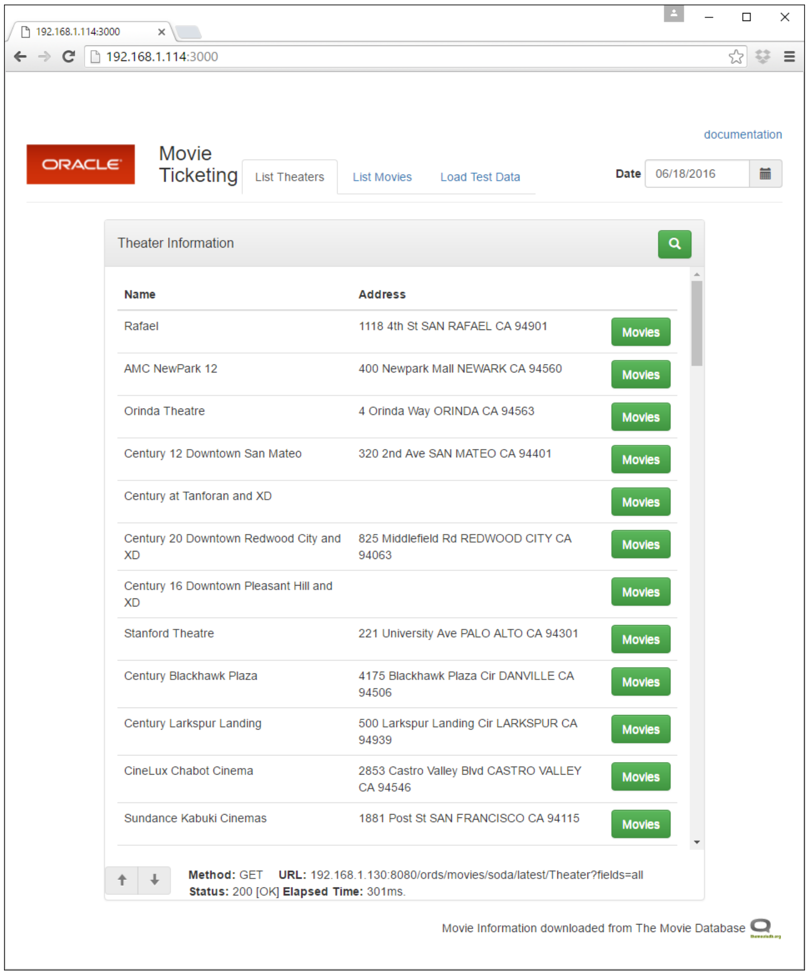
Oracle Movie Ticketing Application (NODE.js with Soda for REST).

This application demonstrates how to develop an application that combines document centric application development techniques, JSON based data persistence, REST Services with an Oracle Database. The application is a simulation of a system for searching movies and theaters and then purchasing tickets to see a given showing of a movie. It is a single-page web application, built using a decoupled AngularJS front-end that communicates with a set of REST services provided by an application tier developed in Node.js. The application tier uses REST to invoke micro-services provided by Oracle's SODA for REST, a component of Oracle Rest Data Services (ORDS). Soda for rest provides data persistence services for JSON and other kinds of document. The application provides the following capabilities:

1. List theaters: Shows the available theaters. Users may drill down to a list of movies showing at the theater by clicking on the "Movies" button associated with a given theater.

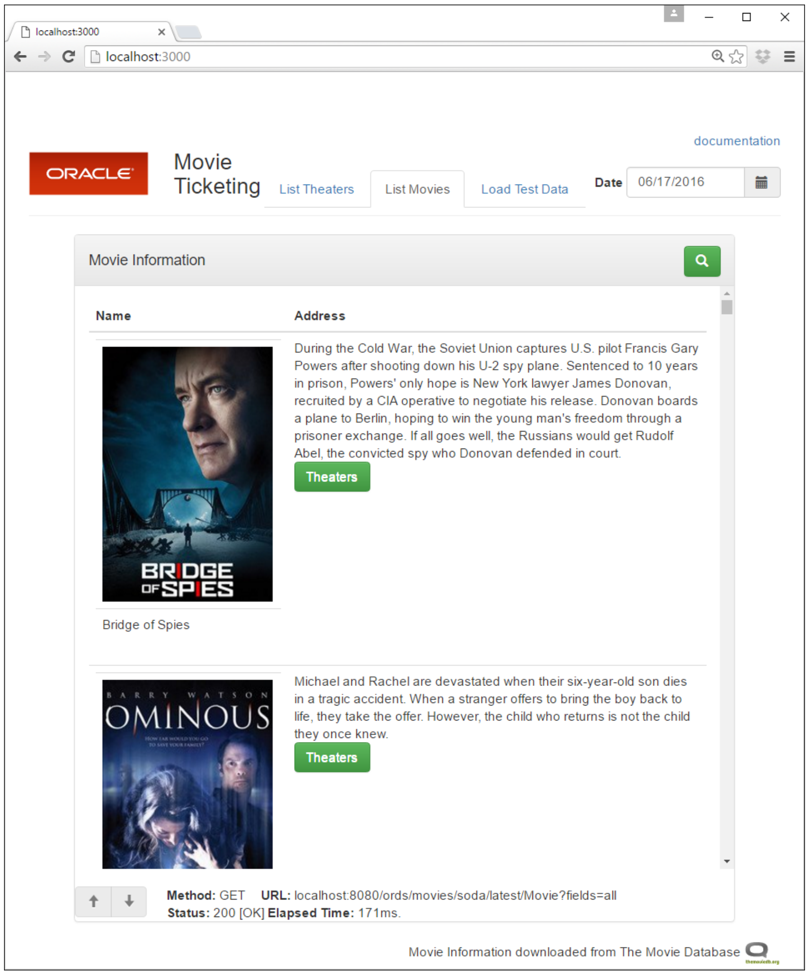


Name	Address	
Rafael	1118 4th St SAN RAFAEL CA 94901	Movies
AMC NewPark 12	400 Newpark Mall NEWARK CA 94560	Movies
Orinda Theatre	4 Orinda Way ORINDA CA 94563	Movies
Century 12 Downtown San Mateo	320 2nd Ave SAN MATEO CA 94401	Movies
Century at Tanforan and XD		Movies
Century 20 Downtown Redwood City and XD	825 Middlefield Rd REDWOOD CITY CA 94063	Movies
Century 16 Downtown Pleasant Hill and XD		Movies
Stanford Theatre	221 University Ave PALO ALTO CA 94301	Movies
Century Blackhawk Plaza	4175 Blackhawk Plaza Cir DANVILLE CA 94506	Movies
Century Larkspur Landing	500 Larkspur Landing Cir LARKSPUR CA 94939	Movies
CineLux Chabot Cinema	2853 Castro Valley Blvd CASTRO VALLEY CA 94546	Movies
Sundance Kabuki Cinemas	1881 Post St SAN FRANCISCO CA 94115	Movies

Method: GET **URL:** 192.168.1.130:8080/ords/movies/soda/latest/Theater?fields=all
Status: 200 [OK] **Elapsed Time:** 301ms.

Movie Information downloaded from The Movie Database

2. List movies: Shows the available movies. Users may drill down to a list of the theaters showing a particular movie by clicking on the "Theaters" button associated with a movie.



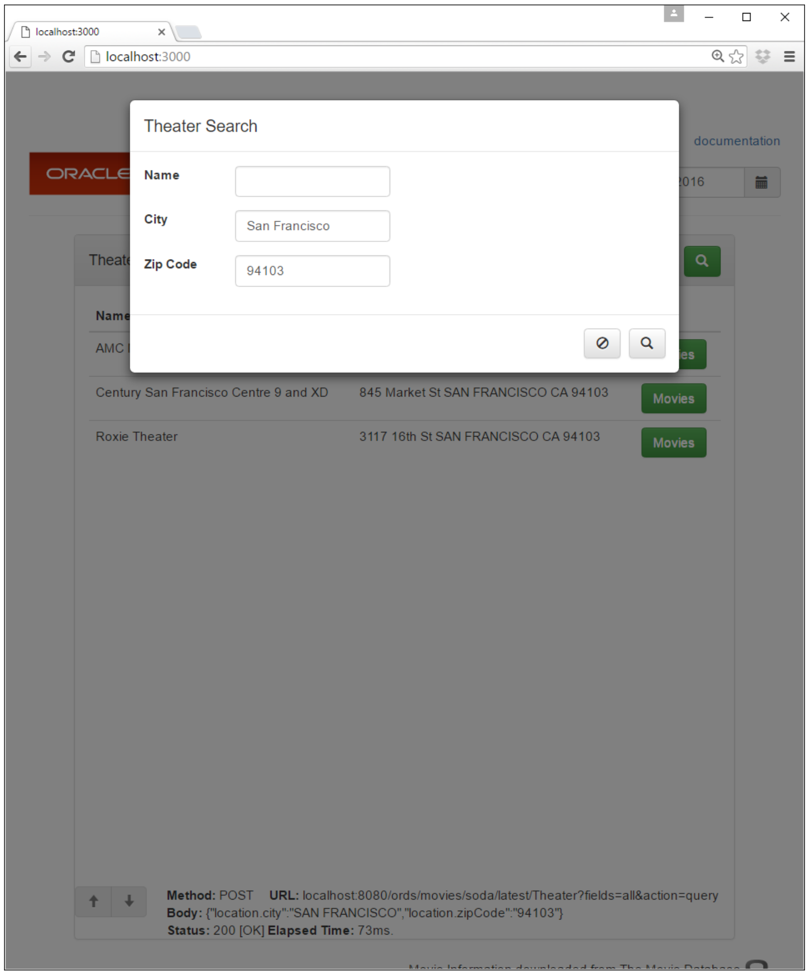
During the Cold War, the Soviet Union captures U.S. pilot Francis Gary Powers after shooting down his U-2 spy plane. Sentenced to 10 years in prison, Powers' only hope is New York lawyer James Donovan, recruited by a CIA operative to negotiate his release. Donovan boards a plane to Berlin, hoping to win the young man's freedom through a prisoner exchange. If all goes well, the Russians would get Rudolf Abel, the convicted spy who Donovan defended in court.

Bridge of Spies

Michael and Rachel are devastated when their six-year-old son dies in a tragic accident. When a stranger offers to bring the boy back to life, they take the offer. However, the child who returns is not the child they once knew.

Method: GET **URL:** localhost:8080/ords/movies/soda/latest/Movie?fields=all
Status: 200 [OK] **Elapsed Time:** 171ms.

3. Search theater by name, city and zip.



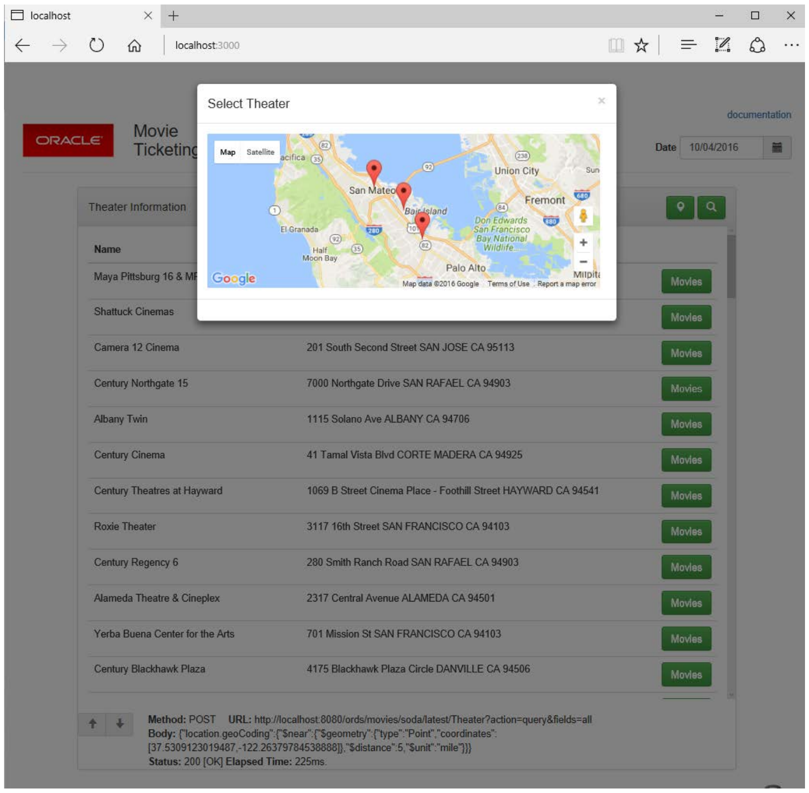
Theater Search

Name	
City	San Francisco
Zip Code	94103

Name		
AMC		
Century San Francisco Centre 9 and XD	845 Market St SAN FRANCISCO CA 94103	Movies
Roxie Theater	3117 16th St SAN FRANCISCO CA 94103	Movies

```
Method: POST   URL: localhost:8080/ords/movies/soda/latest/Theater?fields=all&action=query
Body: {"location.city":"SAN FRANCISCO","location.zipCode":"94103"}
Status: 200 [OK] Elapsed Time: 73ms.
```

4. Search theaters by proximity to current location.

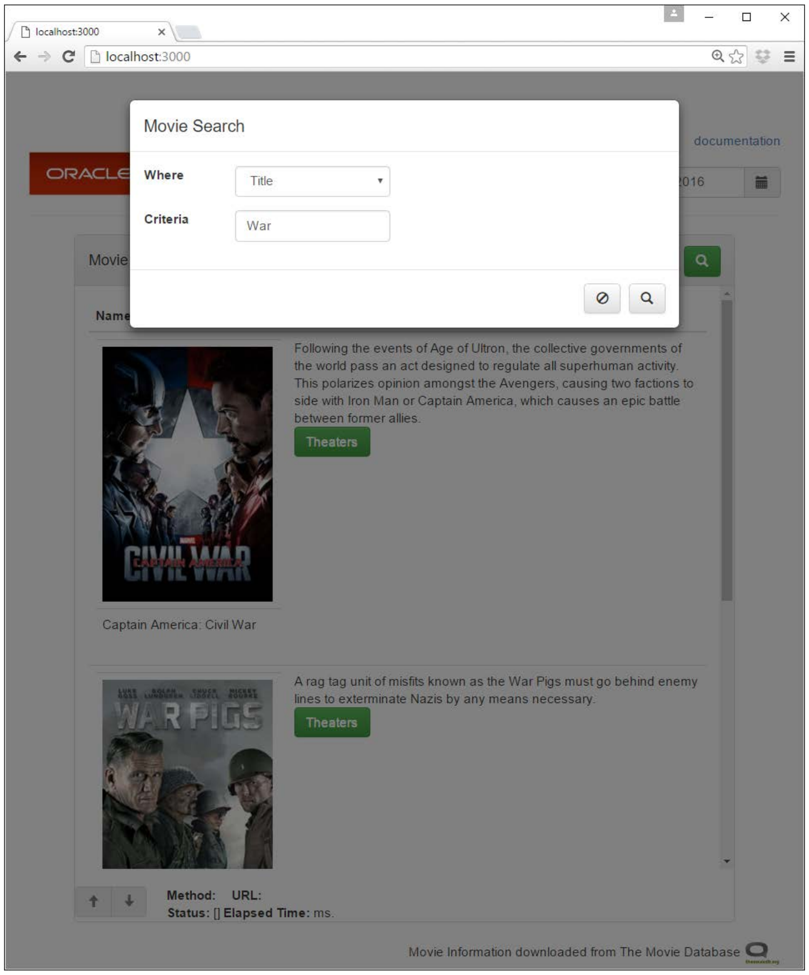


Select Theater

Name		
Maya Pittsburg 16 & MP		Movies
Shattuck Cinemas		Movies
Camera 12 Cinema	201 South Second Street SAN JOSE CA 95113	Movies
Century Northgate 15	7000 Northgate Drive SAN RAFAEL CA 94903	Movies
Albany Twin	1115 Solano Ave ALBANY CA 94706	Movies
Century Cinema	41 Tamal Vista Blvd CORTE MADERA CA 94925	Movies
Century Theatres at Hayward	1069 B Street Cinema Place - Foothill Street HAYWARD CA 94541	Movies
Roxie Theater	3117 16th Street SAN FRANCISCO CA 94103	Movies
Century Regency 6	280 Smith Ranch Road SAN RAFAEL CA 94903	Movies
Alameda Theatre & Cineplex	2317 Central Avenue ALAMEDA CA 94501	Movies
Yerba Buena Center for the Arts	701 Mission St SAN FRANCISCO CA 94103	Movies
Century Blackhawk Plaza	4175 Blackhawk Plaza Circle DANVILLE CA 94506	Movies

```
Method: POST   URL: http://localhost:8080/ords/movies/soda/latest/Theater?action=query&fields=all
Body: {"location.geoCoding":{"$near":{"$geometry":{"type":"Point","coordinates":
[37.5309123019487,-122.26379784538888]},"$distance":5,"$unit":"mile"}}}
Status: 200 [OK] Elapsed Time: 225ms.
```

5. Search movies by title and plot.


Movie Search

Where: Title

Criteria: War

Following the events of Age of Ultron, the collective governments of the world pass an act designed to regulate all superhuman activity. This polarizes opinion amongst the Avengers, causing two factions to side with Iron Man or Captain America, which causes an epic battle between former allies.

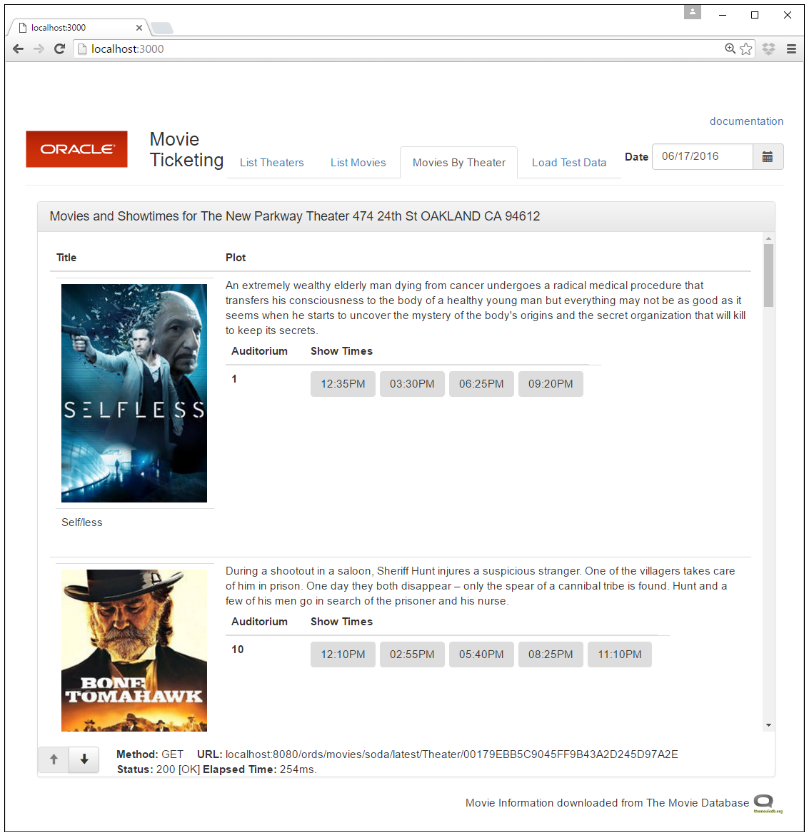
Captain America: Civil War

A rag tag unit of misfits known as the War Pigs must go behind enemy lines to exterminate Nazis by any means necessary.

Method: **URL:**
Status: [] **Elapsed Time:** ms.

Movie Information downloaded from The Movie Database

6. Show the movies playing at a particular theater on a given date. Users can book a ticket to a particular screening by clicking on the show time.



Movies and Showtimes for The New Parkway Theater 474 24th St OAKLAND CA 94612

Title	Plot
Self/less	An extremely wealthy elderly man dying from cancer undergoes a radical medical procedure that transfers his consciousness to the body of a healthy young man but everything may not be as good as it seems when he starts to uncover the mystery of the body's origins and the secret organization that will kill to keep its secrets.

Auditorium	Show Times
1	12:35PM 03:30PM 06:25PM 09:20PM

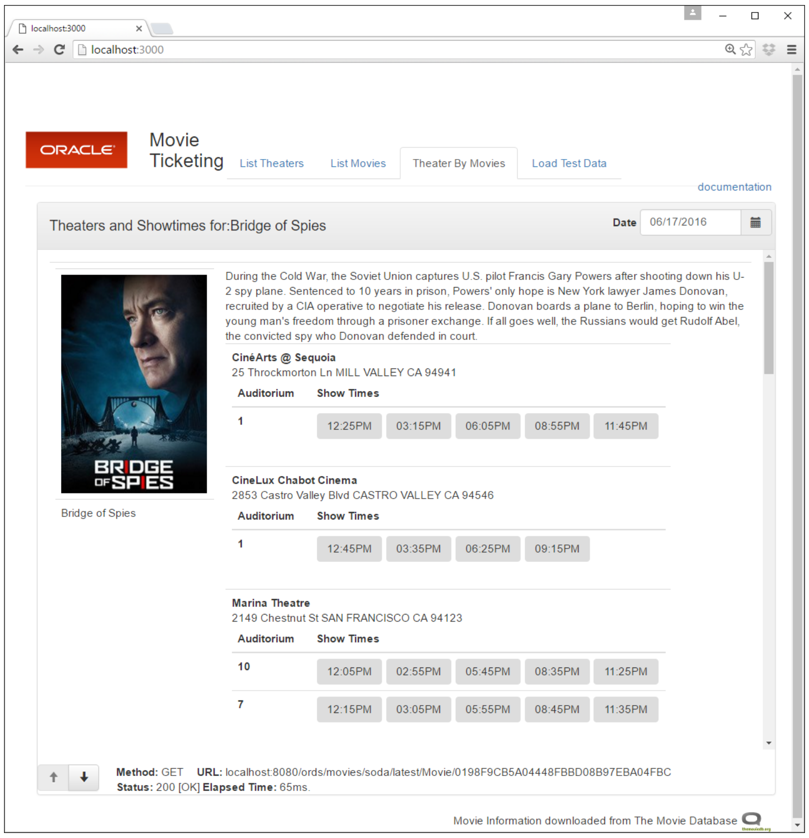
During a shootout in a saloon, Sheriff Hunt injures a suspicious stranger. One of the villagers takes care of him in prison. One day they both disappear – only the spear of a cannibal tribe is found. Hunt and a few of his men go in search of the prisoner and his nurse.

Auditorium	Show Times
10	12:10PM 02:55PM 05:40PM 08:25PM 11:10PM

Method: GET **URL:** localhost:8080/ords/movies/soda/latest/Theater/00179EBB5C9045FF9B43A2D245D97A2E
Status: 200 [OK] **Elapsed Time:** 254ms.

Movie Information downloaded from The Movie Database

7. Show the theaters showing a particular movie on a given date. Users can book a ticket to a particular screening by clicking on the show time.



Theaters and Showtimes for:Bridge of Spies

Bridge of Spies

During the Cold War, the Soviet Union captures U.S. pilot Francis Gary Powers after shooting down his U-2 spy plane. Sentenced to 10 years in prison, Powers' only hope is New York lawyer James Donovan, recruited by a CIA operative to negotiate his release. Donovan boards a plane to Berlin, hoping to win the young man's freedom through a prisoner exchange. If all goes well, the Russians would get Rudolf Abel, the convicted spy who Donovan defended in court.

CinéArts @ Sequoia
25 Throckmorton Ln MILL VALLEY CA 94941

Auditorium	Show Times
1	12:25PM 03:15PM 06:05PM 08:55PM 11:45PM

CineLux Chabot Cinema
2853 Castro Valley Blvd CASTRO VALLEY CA 94546

Auditorium	Show Times
1	12:45PM 03:35PM 06:25PM 09:15PM

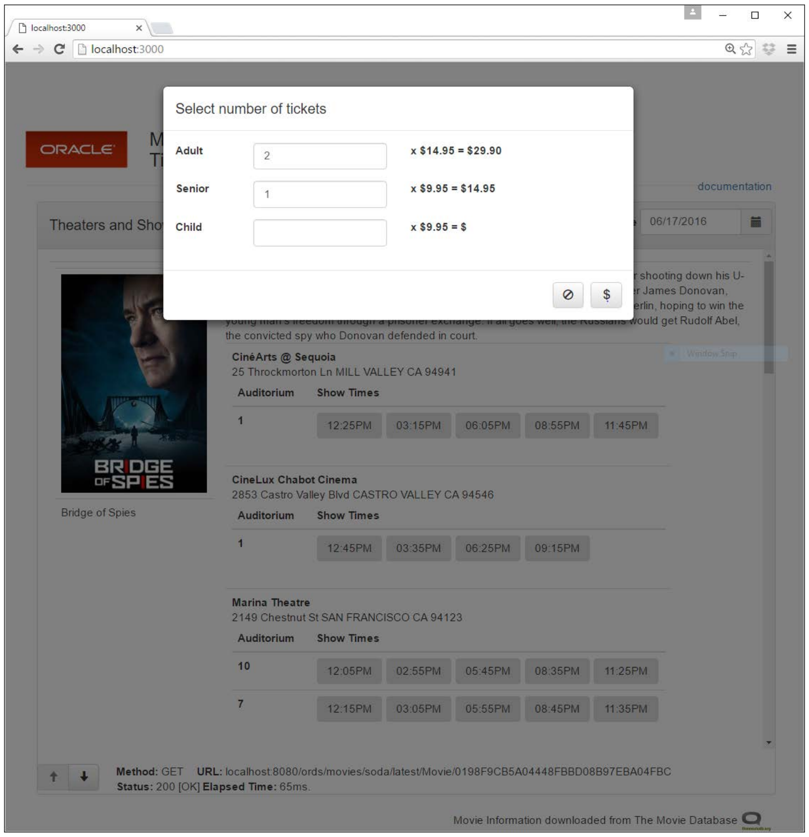
Marina Theatre
2149 Chestnut St SAN FRANCISCO CA 94123

Auditorium	Show Times
10	12:05PM 02:55PM 05:45PM 08:35PM 11:25PM
7	12:15PM 03:05PM 05:55PM 08:45PM 11:35PM

Method: GET **URL:** localhost:8080/ords/movies/soda/latest/Movie/0198F9CB5A04448FBBD08B97EBA04FBC
Status: 200 [OK] **Elapsed Time:** 65ms.

Movie Information downloaded from The Movie Database

8. Purchase tickets to see a particular showing of a movie at a particular theater.



Select number of tickets

Adult	2	x $14.95 = $29.90
Senior	1	x $9.95 = $14.95
Child		x $9.95 = $

r shooting down his U-
r James Donovan,
erlin, hoping to win the
would get Rudolf Abel,
the convicted spy who Donovan defended in court.

Bridge of Spies

CinéArts @ Sequoia
25 Throckmorton Ln MILL VALLEY CA 94941

Auditorium	Show Times				
1	12:25PM	03:15PM	06:05PM	08:55PM	11:45PM

CineLux Chabot Cinema
2853 Castro Valley Blvd CASTRO VALLEY CA 94546

Auditorium	Show Times			
1	12:45PM	03:35PM	06:25PM	09:15PM

Marina Theatre
2149 Chestnut St SAN FRANCISCO CA 94123

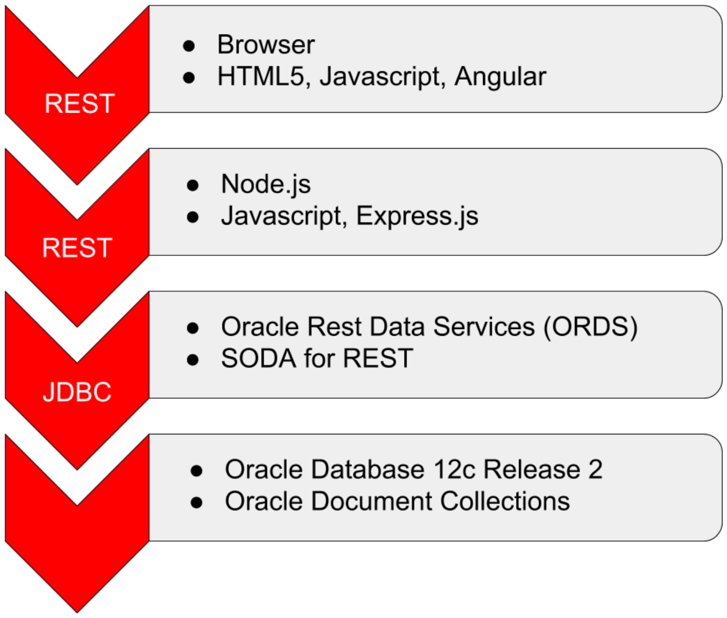
Auditorium	Show Times				
10	12:05PM	02:55PM	05:45PM	08:35PM	11:25PM
7	12:15PM	03:05PM	05:55PM	08:45PM	11:35PM

Method: GET **URL:** localhost:8080/ords/movies/soda/latest/Movie/0198F9CB5A04448FBBD08B97EBA04FBC
Status: 200 [OK] **Elapsed Time:** 65ms.

Movie Information downloaded from The Movie Database

Architecture

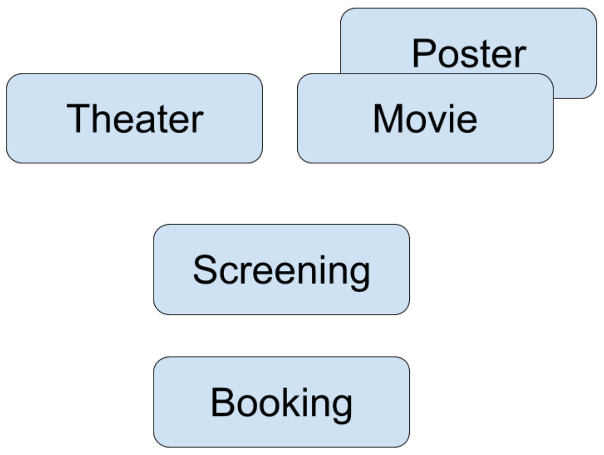
The architecture for this version of the application is as follows:


REST
REST
JDBC
Browser
HTML5, Javascript, Angular
Node.js
Javascript, Express.js
Oracle Rest Data Services (ORDS)
SODA for REST
Oracle Database 12c Release 2
Oracle Document Collections

The front end application uses HTML5 and Angular to communicate with an application-tier developed using JavaScript and Node.js. The application tier makes use of a number of standard node modules, including express and http. The Application tier uses SODA for REST and Oracle REST Data Services (ORDS) to communicate with the back-end data store. The back-end data store is an Oracle 12c Database.

Data Model

The data model for the application is shown in the following diagram:



It consists of four collections, Movie, Theater, Screening and TicketSale that are used to manage the JSON document types that represent the objects used by the application. The 5th collection, Poster, contains Binary Content (Images). Each of the collections has a unique, system generated primary key. Many of the object types also contain an application supplied unique id as part of the object type.

- **Theater**: contains information about a Theater, including a unique id, its location and default information about the number and capacity of its auditoriums. Click the right arrow below to see an sample theater document:

```
{
  "id" : 12,
  "name" : "Orinda Theatre",
  "location" : {
    "street" : "4 Orinda Way",
    "city" : "ORINDA",
    "zipCode" : "94563",
    "state" : "CA",
    "phoneNumber" : null,
    "geoCoding" : {
      "type" : "Point",
      "coordinates" : [-122.19335,37.886116]
    }
  },
  "screens" : [ ... ]
}
```

- **Movie**: contains information about a Movie, including a unique id, title, plot, and cast and crew information. Click the right arrow below to see an sample movie document:

```
{
  "id" : 153518,
  "title" : "The Angry Birds Movie",
    "plot" : "An island populated entirely by happy, flightless birds or almost entirely. In this paradise, Red, a bird with a temper problem, speedy Chuck, and the volatile Bomb have always been outsiders. But when the island is visited by mysterious green piggies, it?s up to these unlikely outcasts to figure out what the pigs are up to.",
    "runtime" : 95,
    "posterURL" : "/movieticket/poster/42ADAE57A4F44BDBAC41925E726673DA",
  ▾ "castMember" : [
    ▾ {
          "name" : "Jason Sudeikis",
          "character" : "Red (voice)"
      },
    ▸ { ... }
    ▸ { ... }
    ▸ { ... }
    ▸ { ... }
    ▸ { ... }
    ▸ { ... }
    ▸ { ... }
    ▸ { ... }
    ▸ { ... }
    ▸ { ... }
    ▸ { ... }
    ▸ { ... }
    ▸ { ... }
    ▸ { ... }
    ▸ { ... }
    ▸ { ... }
    ▸ { ... }
    ▸ { ... }
    ▸ { ... }
    ▸ { ... }
    ▸ { ... }
    ▸ { ... }
    ▸ { ... }
    ▸ { ... }
    ▸ { ... }
    ▸ { ... }
    ▸ { ... }
    ▸ { ... }
    ▸ { ... }
    ▸ { ... }
    ],
  ▸ "crewMember" : [ ... ]
    "releaseDate" : "2016-05-20",
    "certification" : "PG",
    "externalURL" : "http://image.tmdb.org/t/p/w185/t2mZzQXjpQxmqtJOPpe8Dr2YpMl.jpg?api_key=ebdef9b4d4764fa65c16039d09eb0eed"
  }
```

- **Screening**: Details movie showings by theater, screen, start time, ticket pricing for the show and number of seats remaining. Click the right arrow below to see an sample screening document:

```
{
    "theaterId" : 62,
    "movieId" : 382638,
    "screenId" : 1,
    "startTime" : "2016-06-10T20:05:00-07:00",
    "seatsRemaining" : 96,
    "ticketPricing" : {
        "adultPrice" : 14.95,
        "childPrice" : 9.95,
        "seniorPrice" : 9.95
    }
}
```

- **TicketSale**: Contains details of seat purchases. Click the right arrow below to see an sample ticket sale document:

```
{
    "customerId" : 1,
    "adult" : 2,
    "senior" : null,
    "child" : null,
    "adultPrice" : 14.95,
    "seniorPrice" : 9.95,
    "childPrice" : 9.95,
    "startTime" : "2016-06-08T18:35:00-07:00",
    "theaterId" : 12,
    "screenId" : 5,
    "movieId" : 374254,
    "purchaseDate" : "2016-06-08T13:59:59-07:00"
}
```

- **Poster**: Contains binary images of the posters for the movies.

Client Tier

The front end application consists of a single page HTML file named index.html. This file is served up by the Node.js server. The front end application logic, including the AngularJS modules, controllers and factories that invoke the REST services provided by Node.js server are contained in the file movieticketing.js file. The user interface uses the Twitter Bootstrap framework to provide a simple, clean and modern user interface.

Application Tier (Node.js)

The JavaScript code that makes up the application tier is organized into 3 layers.

1. **movie_ticketing.js**: This layer contains the application logic that provides the REST services exposed to the browser based componentry. Some of these services are nothing more that very thin veneers over the underlying collections. Others perform quite complex processing that requires multiple interactions between the application code and the Document Store.

2. **movie_ticket_api.js**: This layer provides a pre-packaged set of operations for each of the Document Types used by the Movie Ticketing application. It is a very thin veneer on top of soda-rest.js that provides collection specific implementations of the functionality exposed by soda-rest.js. This layer is also responsible for managing the collection properties that provide the definition of each of the document collections and indexes used by the application. This information is maintained in the **CollectionProperties.json** document. All operations on collections are routed through this layer.

3. **soda-rest.js**: The sole purpose of this module is to provide a generic handler for the mechanics of invoking SODA for REST operations. This includes setting up the HTTP Request, marshalling any content or arguments that need to be provided in order to make the request and processing and returning the response. This module has no pre-determined knowledge of the collections it is interacting with; it is designed to re-used by any Node.js application that wants to make use of SODA for REST. It provides a set of functions that expose each of the micro-services provided by the SODA for REST API. Each of the functions exposed by this module return a JavaScript Promise, allowing the creation of a Promise based applications. .

In addition to the 3 main modules listed above the application includes the following additional modules:

- **index.js** and **routes.js**: These classes use the popular Express.js framework to handle the incoming request from the client tier and routing them to the appropriate functions exposed by movie_ticketing.js.
- **config.js**: This class manages the connection information required by the application tier in order to talk to ORDS.
- **external_interfaces.js**: This class provides a set of services that are used to load test data from external data sources (Fandango.com and TheMovieDatabase.org).

There are 3 configuration files that are used by the application:

- **config.json**: This file contains the information required to establish a connection between the Node.js layer and the ORDS layer.
- **collections.json**: This file contains the CollectionProperties and Index definitions for the document collections used by the application
- **dataSources.json**: This file provides the connection information for the external websites that are used as a source of test data.

The MovieTicketing application exposes the following Rest services which are consumed by the browser-based front end.

1. **ListTheaters**: List all Theaters
2. **GetTheater**: Get a Theater based on its key
3. **GetTheaterById**: Get a Theater based on its Id.
4. **GetMoviesByTheater**: Get the Movies showing in a Theater on a given date
5. **SearchTheaters**: Search Theaters based on name, city and zip.
6. **ListMovies**: List all Movies
7. **ListMoviesByReleaseDate**: List all Movies, ordered by Release Date
8. **GetMovie**: Get a Movie based on its key
9. **GetMovieById**: Get a Movie based on its Id.
10. **GetTheatersByMovie**: Get the set of Theaters showing a Movie on a given date
11. **SearchMovies**: Search Movies based on Title and Plot
12. **BookTicket**: Book one or more tickets to see a given movie

Some of these services are nothing more that very thin veneers over the underlying collections. Other provide quite extensive processing, requiring multiple interactions between the Application Tier and the Document Management layer.

Overview

When the application is launched the user is presented with 3 tabs, "List Theaters", "List Movies" and "Load Test Data."

- **List Theaters:** shows information about theaters. The information for each theater is stored as JSON documents in the THEATER collection. There is a "Movies" button associated with each theater Clicking the button will open an additional tab that displays information about the movies that are being screened at the theater on the specified date. Customers can purchase tickets to a particularscreening from this tab.
- **List Movies:** shows information about movies. The information about each movie is stored as JSON documents in the Movie collection. These is a "Theaters" button associated with each movie Clicking the button will open an additional tab that display information about the theaters that are screening that movie on the specified date. Customers can purchase tickets to a particularscreening from this tab.
- **Load Test Data:** Provides access to services that populate the Movie Ticketing document store

The following sections will examine the code behind some of these services and also look at how they are consumed in by the Browser-based component.

List Theaters Service

The List Theaters tab is based on the output generated by the rest service associated with a GET operation on the end point /movieticket/theaters. The AngularJS controller used to invoke this service is shown below

```
app.controller('theatersCtrl',function($scope, $http, $cookies, theaterService) {

  $cookies.put('movieTicketGUID', GUID)

  $scope.theaterService = theaterService;

  $http({
    method: 'GET',
    url: '/movieticket/theaters/',
  }).success(function(data, status, headers) {
    $scope.theaterService.theaters = data;
    var path = '/movieticket/movieticketlog/operationId/'+ headers('X-SODA-LOG-TOKEN')
    $http.get(path).success(function(data, status, headers) {
      $scope.theaterService.logRecord = data
      // console.log(JSON.stringify($scope.theaterService.logRecord));
    });
  });

});
```

A snippet of HTML and AngularJS code that is used to render the output of the service as HTML is shown below:

```
<div class="tab-pane active" id="tab_TheaterList">
        <div id="TheaterList" class="panel panel-default" ng-controller="theatersCtrl">
                <div class="panel-heading">
                        <h3 class="panel-title" style="line-height:35px;">Theater
Information
                        <span class="pull-right">
                                <button class="btn btn-success btn-med"
id="btn_TheaterSearch" type="button" onclick="showTheaterSearch()">
                                                <span class="glyphicon glyphicon-
search"></span>
                                        </button>
                            </span>
                        </h3>
        </div>
        <div class="panel-body" style="height:65vh; overflow: auto;">
                <table class="table table-fixed">
                        <thead>
```

```
                                <tr>
                                        <th>Name</th>
                                        <th>Address</th>
                                  <th></th>
                                </tr>
                        </thead>
                        <tbody>
                                <tr ng-repeat="theater in theaterService.theaters">
                                        <td>{{theater.value.name}}</td>
                                        <td>{{theater.value.location.street}}
{{theater.value.location.city}} {{theater.value.location.state}}
{{theater.value.location.zipCode}}</td>
                                        <td>
                                        <button id="btn_MoviesByTheater" type="button"
class="btn btn-default btn-success" ng-
click="theaterService.getMoviesByTheater(theater.id)">Movies</button>
                                  </td>
                                </tr>
                        </tbody>
                </table>
        </div>
                <div>
                        <br>
                </div>
            <div class="container row" id="log_TheaterList">
                <table>
                        <tbody>
                                <tr>
                                        <td style="padding-right:1em; width:100px;
vertical-align:top;">
                                                <div class="btn-group" role="group">
                                                        <button type="button"
class="btn btn-default btn-med" ng-disabled="true" >
                                                                <span class="glyphicon
glyphicon-arrow-up"></span>
                                                        </button>
                                                        <button type="button"
class="btn btn-default btn-med" ng-disabled="true">
                                                                <span class="glyphicon
glyphicon-arrow-down"></span>
                                                        </button>
                                                    </div>
                                                </td>
                                                <td>
                                                        <table>
                                                                <tbody>
                                                                        <tr>
                                                                                <td>
                                                                        <span
style="margin-right:1em;"><strong>Method:</strong>
{{theaterService.logRecord[0].value.method}}</span>
                                                                        <span
style="margin-right:1em;"><strong>URL:</strong>
{{theaterService.logRecord[0].value.url}}</span>
                                                                <td>
                                                        </tr>
                                                                <tr ng-
if="theaterService.logRecord[0].value.request.body.length > 0">

<td><strong>Body:</strong> {{theaterService.logRecord[0].value.request.body}}</td>
                                                                </tr>
                                                                <tr>

<td><strong>Status:</strong> {{theaterService.logRecord[0].value.response.statusCode}}
[{{theaterService.logRecord[0].value.response.statusText}}] <strong>Elapsed
Time:</strong> {{theaterService.logRecord[0].value.elapsedTime}}ms.</td>
                                                                </tr>
                                                        </tbody>
                                                </table>
                                        </td>
                                </tr>
                        </tbody>
                </table>
         </div>
    </div>
```

```
</div>
```

When a GET operation is performed on the URL /movieticket/theaters the Node.js application invokes the function theatersService provided by the module movie_ticketing.api. The code for this module is shown below:

```
function theatersService(sessionState, response, next) {

  console.log('movieTicketing.theatersService()');

  movieAPI.getTheaters(sessionState).then(function (sodaResponse) {
    response.setHeader('X-SODA-LOG-TOKEN',sessionState.operationId);
    response.json(sodaResponse.json);
    response.end();
  }).catch(function(e){
    next(e);
  });
}
```

Listing the contents of a Collection

The theatersService method uses SODA for REST's "List Collection" micro-service to get the list of theaters. It invokes the function getTheaters provided by the movie_ticket_api.js module and returns the resulting JSON as the repsonse. It also adds an X-SODA-LOG-TOKEN header to the response that provides the client application with a unique identifier that can be used to retrieve the log records associated with the operation. The getTheaters method is a very thin veneer over the generic getColleciton method provided by the module soda-rest.js. The code for both functions is shown below:

```
function getTheaters(sessionState, limit,fields) {

   return sodaRest.getCollection(sessionState, 'Theater',limit,fields)

}

function getCollection(sessionState, collectionName,limit,fields) {

  var moduleId = 'getCollection("' + collectionName + '")';

  var requestOptions = {
    method  : 'GET'
  , uri     : getDocumentStoreURI(collectionName)
  , qs      : addLimitAndFields({},limit,fields)
  , headers : setHeaders()
  , time    : true
  , json    : true
  };

  return generateRequest(moduleId, sessionState, requestOptions);

}
```

The getCollection() method lists the content of the specified collection. The method takes the following parameters:

- **sessionState:** Provides session state, in this case primarily used to manage logging of the SODA for REST operations.
- **collectionName:** The name of the collection
- **limit:** Provides control over the number of documents retuned by the operation. The default is 100 documents
- **fields:** Provides control over whether to return just metadata, just content or both. The default is metadata and content.

The function returns a JavaScript Promise object that, on execution, returns the contents of the specified collection. Under the covers the HTTP request is performed using the Node.js module **request.js**. The application logic is as follows:

- Set up an options object. The options object defines the MEHTOD and URI for the HTTP operation. A getCollection() operation is executed by performing a GET on the URI the collection is bound to. The options objects also specifies any query string parameters or HTTP headers that need to be provided as part of the HTTP request.
- Construct a Promise that uses the request.js moodule to execute the getCollection() operaiton

All of the methods exposed by the soda-rest.js module follow a similar design pattern. They construct an options object that specifies the operation to be performed and then use the function generateRequest() to generate a JavaScript promise that invokes the request.js module to perform the required operation. The Promise is then returned to the calling function. The code to generate the promise is shown below:

```
function generateRequest(moduleId, sessionState, requestOptions) {

  return new Promise(function(resolve, reject) {
      // console.log('Execute Promise: ' + moduleId);
    var logRequest = createLogRequest(sessionState, requestOptions)
    request(requestOptions, function(error, response, body) {
      if (error) {
        reject(getSodaError(moduleId,requestOptions,err));
      }
      else {
        processSodaResponse(moduleId, requestOptions, logRequest, response, body,
resolve, reject);
      }
    }).auth(getConnectionProperties().username, getConnectionProperties().password,
true);
  });
}
```

generateRequest() uses a generic callback function, processSodaResponse() to processes the response to the HTTP request and fulfill the Promise returned by the function. If the REST operation is successful a sodaResponse object is constructed, that includes the HTTP Status, headers and response and this object is passed to the Promise's resolve() callback. It the REST call fails for some reason or if it returns an unexpected HTTP status, then an error object is constructued that provides full details of the request and response and this object is to the Promise's reject() callback. The callback used by generateRequest() is shown below:

```
function processSodaResponse(moduleName, requestOptions, logRequest, sodaResponse,
body, resolve, reject) {

  var response = {
    module         : moduleName
  , requestOptions : requestOptions
  , statusCode     : sodaResponse.statusCode
  , statusText     : http.STATUS_CODES[sodaResponse.statusCode]
  , contentType    : sodaResponse.headers["content-type"]
  , headers        : sodaResponse.headers
  , elapsedTime    : sodaResponse.elapsedTime
  }


  if ((body !== undefined) && (body !== null)) {
    if (response.contentType === "application/json") {
      // console.log('processSodaResponse("' + moduleName + '","' +
response.contentType + '","' + typeof body + '")');
      if (typeof body === 'object') {
        response.json = body
      }
      else {
        try {
          response.json = JSON.parse(body);
        }
        catch (e) {
          response.body = body;
        }
      }
      if ((response.json) && (response.json.items)) {
        response.json = response.json.items;
      }
    }
    else {
      // console.log('processSodaResponse("' + moduleName + '","' +
response.contentType + '","' + Buffer.byteLength(body) + '")');
      response.body = body;
    }
  }

  logResponse(response, logRequest);

  if ((sodaResponse.statusCode === 200) || (sodaResponse.statusCode === 201)) {
    resolve(response);
  }
  else {
    response.cause = new Error()
    reject(new SodaError(response));
  }
}
```

List Movies By Theater Service

The "List Movies by Theater" tab is opened when the user clicks on the "Movies" button associated with one of the theaters on the "List Theaters" tab. It lists the show times for the movies playing at the selected theater on the specified date. The content of this tab is based on the output generated by the rest service associated with a GET operation on the end point /theaters/:id/movies/:date, where :id is the id of the theater and :date is the required date. When a GET operation is performed on the URL the Node.js application invokes the function moviesByTheaterService provided by the module movie_ticketing.api passing the values for id and date. The code for this module is shown below:

```
function moviesByTheaterService(sessionState, response, next, id, dateStr) {

  console.log('movieTicketing.moviesByTheaterService(' + id  + ',' + dateStr + ')');

  movieAPI.getTheater(sessionState, id).then(function (sodaResponse) {
    var theater = sodaResponse.json;
    delete(theater.screens);
    return getMoviesByTheaterAndDate(sessionState,theater,dateStr)
  }).then(function (moviesByTheater) {
    // console.log(JSON.stringify(moviesByTheater))
    response.setHeader('X-SODA-LOG-TOKEN',sessionState.operationId);
    response.json(moviesByTheater);
    response.end();
  }).catch(function(e){
    next(e);
  });
}
```

The moviesByTheaterService method invokes the getTheater function provided by the movie_ticket_api.js module to get the information for the specified theater using its internal key. It then calls the function getMoviesByTheaterAndDate to construct a JSON document that summarizes the movies and show times for the specified theater and date. The resulting JSON forms the response to the request. The code for this function is shown below:

```
function getMoviesByTheaterAndDate(sessionState,theater, date) {

  var moviesByTheater = {
    'theater' : theater,
    'movies' : []
  };

  // console.log('getMoviesByTheaterAndDate(' + theater.id + ',' + date + ')');

  var startDate = new Date(Date.parse(date))
  startDate.setHours(0);
  startDate.setMinutes(0);
  startDate.setSeconds(0);
  startDate.setMilliseconds(0);

  var endDate = new Date(Date.parse(date));
  endDate.setHours(0)
  endDate.setMinutes(0)
  endDate.setSeconds(0)
  endDate.setMilliseconds(0);
  endDate.setDate(endDate.getDate() + 1);

  var qbe = { theaterId : theater.id, startTime : { "$gte" : startDate, "$lt" : endDate
}, "$orderby" : { screenId : 1, startTime : 2}};

  return movieAPI.queryScreenings(sessionState, qbe).then(function(items) {
    return processScreeningsByTheaterAndDate(sessionState,items)
  }).then(function(movies) {
    moviesByTheater.movies = movies;
    return moviesByTheater;
  })
}
```

Searching for Documents in Collection

The getMoviesByTheaterAndDate method uses SODA for REST's "QueryByExample" micro-service to locate the required screening documents. It then uses the queryScreenings function provided by the movie_ticket_api.js module to execute the QBE and fetch the required screening documents from the document store. These documents are then passed to the processScreeningsByTheaterAndDate() function in order to generate the required summary. The processScreeningsByTheaterAndDate function uses a second QBE to fetch information about each of movies being shown. The queryScreenings method is a very thin veneer over the generic queryByExample method provided by the module soda-rest.js. The code for both functions is shown below:

```
function queryScreenings(sessionState, qbe,limit,fields) {

  return sodaRest.queryByExample(sessionState, 'Screening',qbe,limit,fields);

}

function queryByExample(sessionState, collectionName, qbe, limit, fields) {

  var moduleId = 'queryByExample("' + collectionName + '",' + JSON.stringify(qbe) +
')';
  console.log(moduleId);

  var requestOptions = {
    method  : 'POST'
  , uri     : getDocumentStoreURI(collectionName)
  , qs      : addLimitAndFields({action : "query"},limit,fields)
  , json    : qbe
  , time    : true
  };

  return generateRequest(moduleId, sessionState, requestOptions);
}
```

A queryByExample is executed by performing a POST operation on the specified collection, specifying the query string 'action=query'. The QBE specification is supplied as the body of the POST. The function returns a Promise that, on execution, returns the result of evaluating the Query-By-Example operation. The actual application logic is similar to the logic for the getCollection function, with the exception that this function performs a POST rather than a GET and the QBE specification is supplied as the body of the POST.

Purchase Tickets Service

Both the List Movies By Theater and List Theaters by Movie tab allow the user to purchase tickets to see a movie by clicking on the showing they would like to attend. Once they have entered the number of tickets they require they can complete the purchase by clicking on the '$' icon. The purchase is made by performing a POST operation on the URL /movieticket/bookTickets. When a POST operation is performed on this URL the Node.js application invokes the function bookTicketService() provided by the module movie_ticketing.api. The information about the number of tickets required is provided as the body of the POST operation. The code for this module is shown below:

```
function bookTicketService(sessionState, response, next, bookingRequest) {

  // console.log('movieTicketing.bookTicketService(' + JSON.stringify(bookingRequest)
+')');

  bookTickets(sessionState, bookingRequest).then(function (bookingStatus) {
    response.setHeader('X-SODA-LOG-TOKEN',sessionState.operationId);
    response.json(bookingStatus);
    response.end();
  }).catch(function(err) {
    next(err);
  });

}
```

The bookTicketService method invokes the bookTickets function provided by the module movie_ticketing.api to record the ticket sale. This functions returns a Promise that when executed will generate a simple JSON document that indicates whether or not the booking was successful. This document is returned to the front end. The code for this function is shown below:

```
function bookTickets(sessionState, bookingRequest) {

  // console.log('movieTicketing.bookTickets(' + JSON.stringify(bookingRequest) +')');

  var key           = bookingRequest.key;
  var eTag            null;
  var screening     = {}
  var seatsRequired = bookingRequest.adult + bookingRequest.senior +
bookingRequest.child;

  return movieAPI.getScreening(sessionState, key).then(function(sodaResponse) {
        eTag = sodaResponse.eTag;
        screening = sodaResponse.json;
          if (screening.seatsRemaining < seatsRequired) {
        return {
          status : 'SoldOut',
          message : 'Only ' + screening.seatsRemaining + ' seats are available for this
performance.'
      };
    }
    else {
      screening.seatsRemaining = screening.seatsRemaining - seatsRequired;
      return movieAPI.updateScreening(sessionState, key, screening,
eTag).then(function(sodaResponse) {
        switch (sodaResponse.statusCode) {
          case 200: // Seat Reserved : Record Ticket Sale
            var ticketSale = makeTicketSale(bookingRequest, screening);
            return movieAPI.insertTicketSale(sessionState,
ticketSale).then(function(sodaResponse) {
              switch (sodaResponse.statusCode) {
                case 201: // Booking Completed
                  return {
                    status  : "Booked",
                    message : "Please enjoy your movie."
                  }
                default:
                        throw sodaResponse;
              }
            }).catch(function (err) {
                throw err;
            })
          default:
            throw sodaResponse;
        }
      }).catch(function (err) {
        switch (err.statusCode) {
          case 412: // Conflicting Ticket Sales : Try again
            return bookTickets(sessionState,bookingRequest)
          default:
            throw err;
        }
      })
    }
  }).catch(function (err) {
        throw err;
  })
}
```

The application logic for completing the booking is as follows:

1. Calculate the total number of seats required.
2. Get the latest version of the specified screening document using its internal key. Retrieve the document and the associated metadata.
3. Check the number of seats remaining for the required screening
    1. Insufficent seats remain to fulfill the request:
        1. Return a 'Sold Out' message
    2. Sufficient seats remain to fulfull the request:
        1. Decrement the seats count for the screening and complete the booking process
        2. Update the Screening document using the updateScreening method provided by the movie_ticket_api.js module.
        3. Check the status of the HTTP request
            1. Status 200: Successful update
                1. Construct the Ticket Sale document
                2. Insert the ticketSale document into the document using the insertTicketSale method provided by the movie_ticket_api.js module.
                3. Return a 'Booking Successful' message
            2. Status 412 : There has been a conflicting update
                1. Repeat this process

Updating Documents

The bookTickets method uses SODA for REST's "UpdateDocument" micro-service to update the screening document. The update of the screening document is handled by the call to the function updateScreening provided by the movie_ticket_api.js module, which in turn calls the putJSON and putDocument functions provided by the soda-rest.js module. The updateScreening and putJSON functions are simply thin veneers over the putDocument function. The putDocument function performs the REST operation that updates the document. The code for these functions is shown below:

```
function updateScreening(sessionState, key,screening,eTag) {

  return sodaRest.putJSON(sessionState, 'Screening',key,screening,eTag);

}

function putJSON(sessionState, collectionName,key,json,eTag) {

  // console.log('putJSON(' + collectionName + ',' + key + ')');

  var serializedJSON = JSON.stringify(json);
  // console.log(serializedJSON);

  return putDocument(sessionState,
collectionName,key,serializedJSON,'application/json',eTag);

}

function putDocument(sessionState, collectionName, key, document, contentType, eTag) {

  var moduleId = 'putDocument(' + collectionName + '","' + key + '","' + contentType +
'")';

  var requestOptions = {
    method  : 'PUT'
  , uri     : getDocumentStoreURI(collectionName) + '/' + key
  , headers : setHeaders(contentType , eTag)
  , time    : true
  };
  if (contentType === 'application/json') {
        requestOptions.json = document
  }
  else {
    requestOptions.body = document
  }

  return generateRequest(moduleId, sessionState, requestOptions);
}
```

An "Update Document" operation is executed by performing a PUT on the target document. The document is identified using its internal key. The new content for the document is supplied as the body of the PUT. The putDocument function returns a Promise that, on execution, returns the result of update operation. The actual application logic is similar to the logic for the getCollection function, with the exception that this function performs a PUT rather than a GET and the uddated document is supplied as the body of the POST

SODA for REST uses an optimistic locking strategy to prevent conflicting updates from taking place. This follows REST conventions and is based on the value of the documents ETag. Any time an application might need to update a document it must fetch the document metadata as well as document content in order to get the current value of the ETag. Conflicting updates are prevented by adding an "If Match" header to the HTTP request. The value of this header is the ETag that was obtained last time the document was read. If there has been a conflicting update the current ETag for the document will not match the ETag supplied by the application and the PUT operation will return HTTP Status code of 412 [Precondition Failed] indicating that the PUT operation was not successful. If there are no conflicting updates then the PUT operation will return HTTP Status code 200 [Successful].

Creating Documents

The bookTickets method uses SODA for REST's "InsertDocument" micro-service to create the ticketSale document. The creation of the new document is handled by the call to the function insertTicketSale provided by the movie_ticket_api.js module, which in turn calls the postJSON and postDocument functions provided by the soda-rest.js module. The insertTicketSale and postJSON functions are simply thin veneers over the postDocument function. The postDocument function performs the REST operation that inserts the document. The code for these functions is shown below:

```
function insertTicketSale(sessionState, ticketSale) {

  return sodaRest.postJSON(sessionState, 'TicketSale',ticketSale);

}

function postJSON(sessionState, collectionName,json) {

  // console.log('postJSON(' + collectionName + ')');

  var serializedJSON = JSON.stringify(json);
  // console.log(serializedJSON);

  return postDocument(sessionState, collectionName,serializedJSON,'application/json');

}

function postDocument(sessionState, collectionName, document, contentType) {

  var moduleId = 'postDocument("' + collectionName + '","' + contentType + '")';

  var requestOptions = {
    method  : 'POST'
  , uri     : getDocumentStoreURI(collectionName)
  , headers : setHeaders(contentType , undefined)
  , time    : true
  };

  if (contentType === 'application/json') {
    requestOptions.json = document
  }
  else {
    requestOptions.body = document
  }
  return generateRequest(moduleId, sessionState, requestOptions);
}
```

An "Insert Document" operation is executed by performing a POST on the target collection. The content of the new document is supplied as the body of the POST. The postDocument function returns a Promise that, on execution, returns the result of insert operation. The actual application logic is similar to the logic for the getCollection function, with the exception that this function performs a POST rather than a GET and the new document is supplied as the body of the POST

Installation and Configuration

These instructions assume you already have access to an Oracle Database 12.1.0.2.0 instance with Bundle Patch 13 installed. If you do not have this available you can start by downloading the latest version of the Oracle Developer Days VM from the Oracle Technology Network. Although this VM comes with a version of Oracle Rest Data Services installed, you will need to update it from 3.0.4 to 3.0.5 by following step 1 of the instructions below

The MovieTicketing demonstration works with sample data downloaded from publically available websites. The THEATER collection is populated with data obtained from an RSS feed published by Fandango.com. Optionally, this data can be enriched with location information obtained from a geocoding service, such as the one provided by Google. The Movie and Poster collections are populated using data obtained from the website themoviedatabase.org (TMDb). In order to obtain data from TMDb and Google you must register with those sites and obtain an API key.

1. Obtain API keys for TMDb and Google.

1. Register for a Google Geocoding API key by following the instructions found here.
2. Make a note of your Google API key.
3. Register for a themoviedatabase.org (TMDb) account by following the instructiosn found here.
4. Sign in to your account, click the API link in their left hand menu and follow the instructions to request an API key.
5. Make a note of your TMDb API key.

2. Install and Configure Node.js

1. Download Node.js from nodejs.org

2. Install Node.js

   The following commands will work on Enterprise Linux, assuming the node installation tarball is in the users Downloads folder, for other environments please refer to the platform specific platform specific installation instructions at https://docs.npmjs.com/getting-started/installing-node.

```
            $ bash
            $ cd
            $ tar xf Downloads/node-v4.4.5-linux-x64.tar.xz
            $ export PATH=$PATH:~/node-v4.4.5-linux-x64/bin
```

3. Use NPM to install the application and it's dependencies from GitHUB

```
            $ mkdir NodeExample
            $ cd NodeExample
            $ npm install oracle-movie-ticket-demo
```

This should result in output similar to this (note your version numbers may be later that the ones shown below):

```
            oracle-movie-ticket-demo@2.0.7 node_modules\oracle-movie-ticket-demo
            +-- angular-cookies@1.5.8
            +-- angular@1.5.8
            +-- @google/maps@0.2.1
            +-- jquery@2.2.4
            +-- bootstrap@3.3.7
            +-- cookie-parser@1.4.3 (cookie-signature@1.0.6, cookie@0.3.1)
            +-- parse-address@0.0.5 (xregexp@2.0.0)
            +-- morgan@1.7.0 (on-headers@1.0.1, basic-auth@1.0.4, depd@1.1.0, on-
finished@2.3.0, debug@2.2.0)
            +-- express-session@1.14.1 (cookie-signature@1.0.6, cookie@0.3.1,
parseurl@1.3.1, utils-merge@1.0.0, on-headers@1.0.1, depd@1.1.0, crc@3.4.0, debug@2.2.0,
                uid-safe@2.1.2)
            +-- body-parser@1.15.2 (content-type@1.0.2, bytes@2.4.0, depd@1.1.0,
qs@6.2.0, on-finished@2.3.0, raw-body@2.1.7, debug@2.2.0, iconv-lite@0.4.13, type-
is@1.6.13,
                http-errors@1.5.0)
            +-- serve-static@1.11.1 (escape-html@1.0.3, encodeurl@1.0.1, parseurl@1.3.1,
send@0.14.1)
            +-- express@4.14.0 (array-flatten@1.1.1, escape-html@1.0.3, utils-
merge@1.0.0, cookie-signature@1.0.6, parseurl@1.3.1, encodeurl@1.0.1, methods@1.1.2,
                merge-descriptors@1.0.1, content-type@1.0.2, cookie@0.3.1, etag@1.7.0,
vary@1.1.0, fresh@0.3.0, path-to-regexp@0.1.7, range-parser@1.2.0,
                content-disposition@0.5.1, depd@1.1.0, qs@6.2.0, on-finished@2.3.0,
debug@2.2.0, finalhandler@0.5.0, type-is@1.6.13, proxy-addr@1.1.2, accepts@1.3.3,
                send@0.14.1)
            +-- bootstrap-datepicker@1.6.4
            +-- request@2.75.0 (tunnel-agent@0.4.3, aws-sign2@0.6.0, forever-
agent@0.6.1, oauth-sign@0.8.2, is-typedarray@1.0.0, caseless@0.11.0, stringstream@0.0.5,
                isstream@0.1.2, aws4@1.4.1, json-stringify-safe@5.0.1, extend@3.0.0,
tough-cookie@2.3.1, qs@6.2.1, node-uuid@1.4.7, combined-stream@1.0.5, mime-types@2.1.12,
                form-data@2.0.0, hawk@3.1.3, bl@1.1.2, http-signature@1.1.1, har-
validator@2.0.6)
            +-- xml2js@0.4.17 (sax@1.2.1, xmlbuilder@4.2.1)
```

At this point the Node.js server has been installed and the application has been downloaded and is ready to run

3. Install, configure and start Oracle Rest Data Services (ORDS)

1. Download the latest version of ORDS from the Oracle Technology Network website

2. Install ORDS

The following commands will work on Enterprise Linux, assuming the node installation zip file is in the users Downloads folder, for other environments please refer to the platform specific installation instructions at https://docs.oracle.com/cd/E37099_01/doc.20/e25066/toc.htm.

```
            $ cd
            $ mkdir ORDS
            $ cd ORDS
            $ unzip ../Downloads/ords.3.0.5.124.10.54.zip
```

Make sure that the database and listener are started and you know a TNS Alias that can connect to the database where ORDS will be installed. Then start the ORDS configuration process as shown below

```
            $ java -jar ords.war
            This Oracle REST Data Services instance has not yet been configured.
            Please complete the following prompts
            Enter the location to store configuration data:/home/oracle/ORDS/config
            Enter the name of the database server [localhost]:
            Enter the database listen port [1521]:
            Enter 1 to specify the database service name, or 2 to specify the database
SID [1]:1
            Enter the database service name:ORCL
            Enter the database password for ORDS_PUBLIC_USER:
            Confirm password:
            Please login with SYSDBA privileges to verify Oracle REST Data Services
schema.
            Enter the username with SYSDBA privileges to verify the installation
[SYS]:sys
            Enter the database password for sys:
            Confirm password:
            Oracle REST Data Services will be installed in ORCL
            Enter 1 if you want to use PL/SQL Gateway or 2 to skip this step.
            If using Oracle Application Express or migrating from mod_plsql then you
must enter 1 [1]:2
```

At this point ORDS will be installed into the target database. The output will be as follows:

```
            Jun 17, 2016 6:18:24 PM
oracle.dbtools.common.config.file.ConfigurationFilesBase update
            INFO: Updated configurations: defaults, apex_pu
            Installing Oracle REST Data Services version 3.0.5.124.10.54
            ... Log file written to
/home/oracle/ORDS/logs/ords_install_core_2016-06-17_181824_00306.log
            ... Verified database prerequisites
            ... Created Oracle REST Data Services schema
            ... Created Oracle REST Data Services proxy user
            ... Granted privileges to Oracle REST Data Services
            ... Created Oracle REST Data Services database objects
            Completed installation for Oracle REST Data Services version
3.0.5.124.10.54. Elapsed time: 00:00:14.299
            Enter 1 if you wish to start in standalone mode or 2 to exit [1]:2
```

ORDS is now installed

3. Create an ORDS user that has permissions to use SODA for REST

```
            java -jar ords.war user MovieTicketing "SODA Developer"
            Enter a password for user MovieTicketing:
            Confirm password for user MovieTicketing:
            Jun 17, 2016 6:29:33 PM oracle.dbtools.standalone.ModifyUser execute
            INFO: Created user: MovieTicketing in file:
/home/oracle/ORDS/config/ords/credentials
```

4. Increase the size of the ORDS JDBC connection pool

Edit the file defaults.xml. Assuming you specified /home/oracle/ORDS/config when responding to the prompt "Enter the location to store configuration data:" in step 2.1 the file is located in the folder config/ords. Locate the entry with the key "jdbc.MaxLimit" and change the value to 500. Locate the entry with the key "jdbc.InitialLimit" and change the value to 50. Save the file

5. Start the ORDS Server

   The first time you start the ORDS server you are asked whether you want to use HTTP or HTTPS and to select the port the server is to listen on. This information is only required the first time ORDS is started.

```
            [oracle@localhost ORDS]$ java -jar ords.war standalone
            Enter 1 if using HTTP or 2 if using HTTPS [1]:1
            Enter the HTTP port [8080]:
            2016-06-17 18:22:54.846:INFO::main: Logging initialized @5506ms
            Jun 17, 2016 6:22:55 PM oracle.dbtools.standalone.StandaloneJetty
setupDocRoot
            INFO: Disabling document root because the specified folder does not exist:
/home/oracle/ORDS/config/ords/standalone/doc_root
            2016-06-17 18:22:55.994:INFO:oejs.Server:main: jetty-9.2.z-SNAPSHOT
            Jun 17, 2016 6:22:56 PM oracle.dbtools.auth.crypto.CryptoKeysGenerator
startup
            INFO: No encryption key found in configuration, generating key
            Jun 17, 2016 6:22:56 PM oracle.dbtools.auth.crypto.CryptoKeysGenerator
startup
            INFO: No mac key found in configuration, generating key
            Jun 17, 2016 6:22:56 PM
oracle.dbtools.common.config.file.ConfigurationFilesBase update
            INFO: Updated configurations: defaults
            Jun 17, 2016 6:22:56 PM oracle.dbtools.auth.crypto.CryptoKeysGenerator
startup
            INFO: Updated configuration with generated keys
            2016-06-17 18:22:56.539:INFO:/ords:main: INFO: Using configuration folder:
/home/oracle/ORDS/config/ords
            2016-06-17 18:22:56.539:INFO:/ords:main: FINEST: |ApplicationContext
[configurationFolder=/home/oracle/ORDS/config/ords, services=Application Scope]|
            Jun 17, 2016 6:22:56 PM oracle.dbtools.common.config.db.DatabasePools
validatePool
            INFO: Validating pool: |apex|pu|
            Jun 17, 2016 6:22:57 PM oracle.dbtools.common.config.db.DatabasePools
validatePool
            INFO: Pool: |apex|pu| is correctly configured
            config.dir
            2016-06-17 18:22:57.381:INFO:/ords:main: INFO: Oracle REST Data Services
initialized|Oracle REST Data Services version : 3.0.5.124.10.54|Oracle REST Data
Services server info: jetty/9.2.z-SNAPSHOT|
            2016-06-17 18:22:57.384:INFO:oejsh.ContextHandler:main: Started
o.e.j.s.ServletContextHandler@2d209079{/ords,null,AVAILABLE}
            2016-06-17 18:22:57.410:INFO:oejs.ServerConnector:main: Started
ServerConnector@2c8d66b2{HTTP/1.1}{0.0.0.0:8080}
            2016-06-17 18:22:57.414:INFO:oejs.Server:main: Started @8085ms
            ^C2016-06-17 18:28:39.476:INFO:oejs.ServerConnector:Thread-1: Stopped
ServerConnector@2c8d66b2{HTTP/1.1}{0.0.0.0:8080}
            2016-06-17 18:28:39.482:INFO:oejsh.ContextHandler:Thread-1: Stopped
o.e.j.s.ServletContextHandler@2d209079{/ords,null,UNAVAILABLE}
```

   At this point the ORDS server is running and ready to service SODA for REST operations

4. Create the Database Schema that will manage the MovieTickets document collections

1. Connect to the database and execute the following commands

```
            $ sqlplus system@ORCL

            SQL*Plus: Release 12.1.0.2.0 Production on Fri Jun 17 19:01:37 2016
            Copyright (c) 1982, 2016, Oracle.  All rights reserved.

            Enter password:
```

```
            Last Successful login time: Fri Jun 17 2016 18:14:03  07:00

            Connected to:
            Oracle Database 12c Enterprise Edition Release 12.1.0.2.0   64bit Production

            SQL> grant connect, resource, unlimited tablespace, SODA_APP to MOVIES
identified by MOVIES;

            Grant succeeded.

            SQL> connect MOVIES/MOVIES@ORCL
            Connected.
            SQL> begin
            2    ORDS.enable_schema();
            3    commit;
            4  end;
            5  /

            PL/SQL procedure successfully completed.

            SQL> quit
            Disconnected from Oracle Database 12c Enterprise Edition Release 12.1.0.2.0
  64bit Production
```

The SODA_APP role is required to use Oracle Document Collections. The call to ORDS.enable_schema() is required to allow the schema to be accessed via ORDS. Both steps are required to use SODA for REST.

5. Update configuration files.

1. config.json

   The config.json file supplies the Node.js server with the information required to connect to the ORDS server. This file must be updated with the username and password and path to endpoint for the Movie Ticketing document store. Click the right arrow below to see a sample config.json document:

   ▸ { ... }

   The file is located in the folder NodeExample/node_modules/oracle-movie-ticket-sample The keys in this file are self explanatory

   - **hostname:** The name or ipaddress of the machine hosting the ORDS instance
   - **port:** The Port number the ORDS instance in listening on. This was specified the first time the ORDS instance was started (See step 3.4).
   - **path:** The endpoint that provides access to the Movie Ticketing document store. The second component of this path is the database schema name (See step 3.1). The schema name is entered lowercase.
   - **username & password** The user name and password used to authenticate with the ORDS server. Note this is the ORDS username and password from step 3.3, not the database schema name and password from step 4.1.

2. dataSources.json

   If the application needs use a proxy server to connect to these sites information about the proxy server must be added to the dataSources.json file.

   Click the right arrow below to see a sample dataSources.json document:

▸ { ... }

The file is located in the folder NodeExample/node_modules/oracle-movie-ticket-sample. The following keys may need to be edited before running the application.

- **useProxy:** Set to true if you need to use a proxy server to access external web sites.
- **proxy.hostname:** If a proxy server is required enter the hostname or ipaddress of the proxy server
- **proxy.port:** If a proxy server is required enter the port number used to communicate with the proxy server
- **tmdb.searchCriteria:**The application loads a small subset of the movies available on TMDb. Movies are selected based on a number of criteria, including release date and rating. To change which the movies selected update these fields before loading movie information.
- **fandango.searchCriteria.zipCode:** The zip code to use when obtaining a list of nearby theaters from Fandango

6. Start the application servers

1. Start Node.js server.

The following commands will work on Enterprise Linux, assuming you followed the steps outlined above.

```
$ cd
$ export PATH=$PATH:~/node-v4.4.5-linux-x64/bin
$ cd NodeExample/oracle-movie-ticket-demo
$ node index.js
```

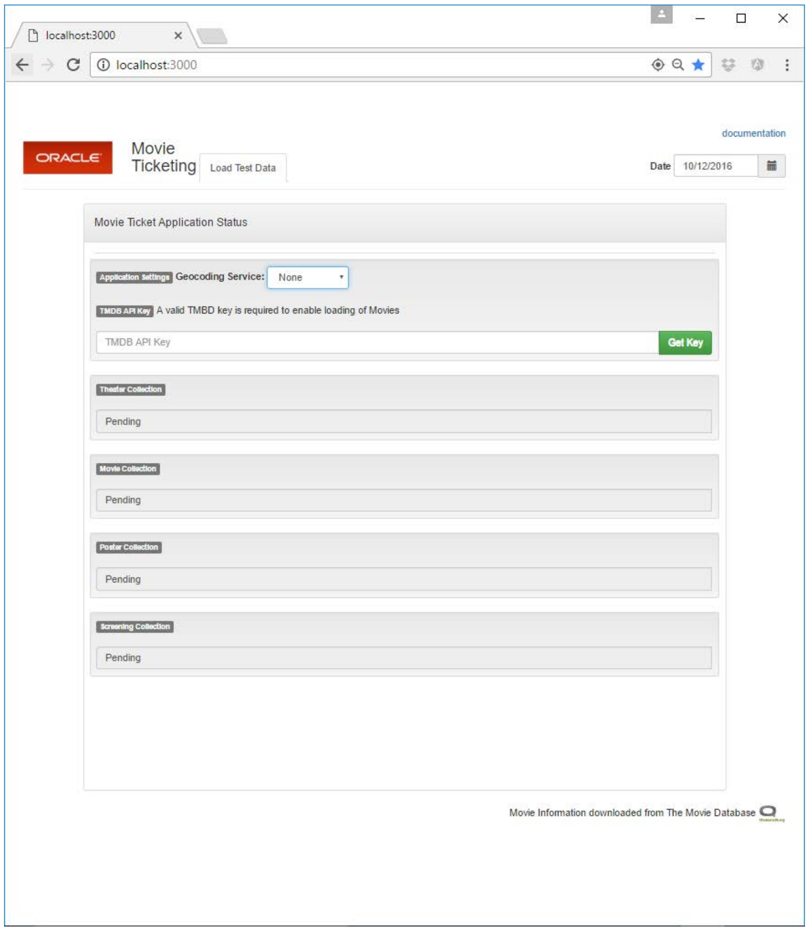
And the system should respond with something similar to this

```
2016-10-12T13:34:52.819Z: MovieTicket Webserver listening on localhost:3000
2016-10-12T13:34:53.235Z: $contains operator supported:  true
2016-10-12T13:34:53.235Z: $near operatator   supported:  true
```

7. Launch the application using a Browser.

1. Navigate to the application start page. The application is configured to run on Port 3000.

The first time you run the application you should see a page like this:



localhost:3000

documentation

Movie Ticketing

Load Test Data

Date 10/12/2016

Movie Ticket Application Status

Application Settings Geocoding Service: None

TMDB API Key A valid TMBD key is required to enable loading of Movies

TMDB API Key

Get Key

Theater Collection

Pending

Movie Collection

Pending

Poster Collection

Pending

Screening Collection

Pending

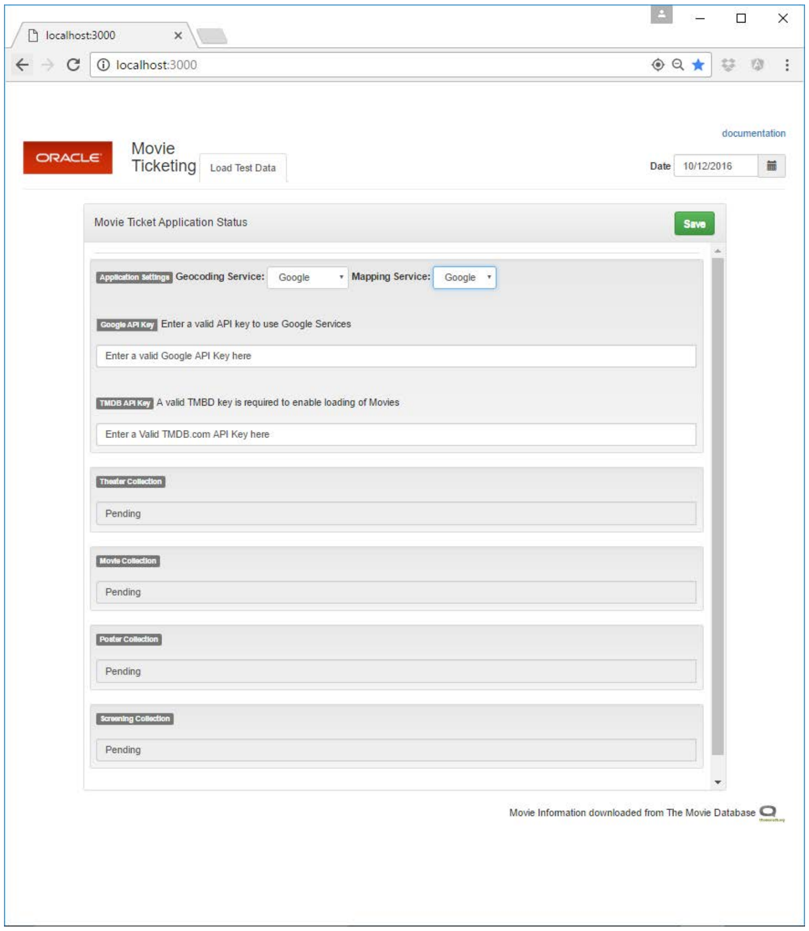
Movie Information downloaded from The Movie Database

This page is used configure the application and obtain the data required to run the application.

2. Configure Application

Movie Information is loaded using services provided by the website themoviedb.org. Theater information is obtained using an RSS feed published by Fandango.com.

If the target document store supports Oracle Location services and the version of Query-By-Example included with SODA supports the $near operator you are given option of geocoding the Theater information. Currently, only the US Government Census Bureau's Geocoding service and Google's Geocoding service are supported. If Geocoding is enabled you are given the option viewing theater locality graphically using a Mapping service. Currently only Google's Mapping service is supported.



Movie Ticket Application Status

Application Settings Geocoding Service: Google Mapping Service: Google

Google API Key Enter a valid API key to use Google Services

Enter a valid Google API Key here

TMDB API Key A valid TMBD key is required to enable loading of Movies

Enter a Valid TMDB.com API Key here

Theater Collection

Pending

Movie Collection

Pending

Poster Collection

Pending

Screening Collection

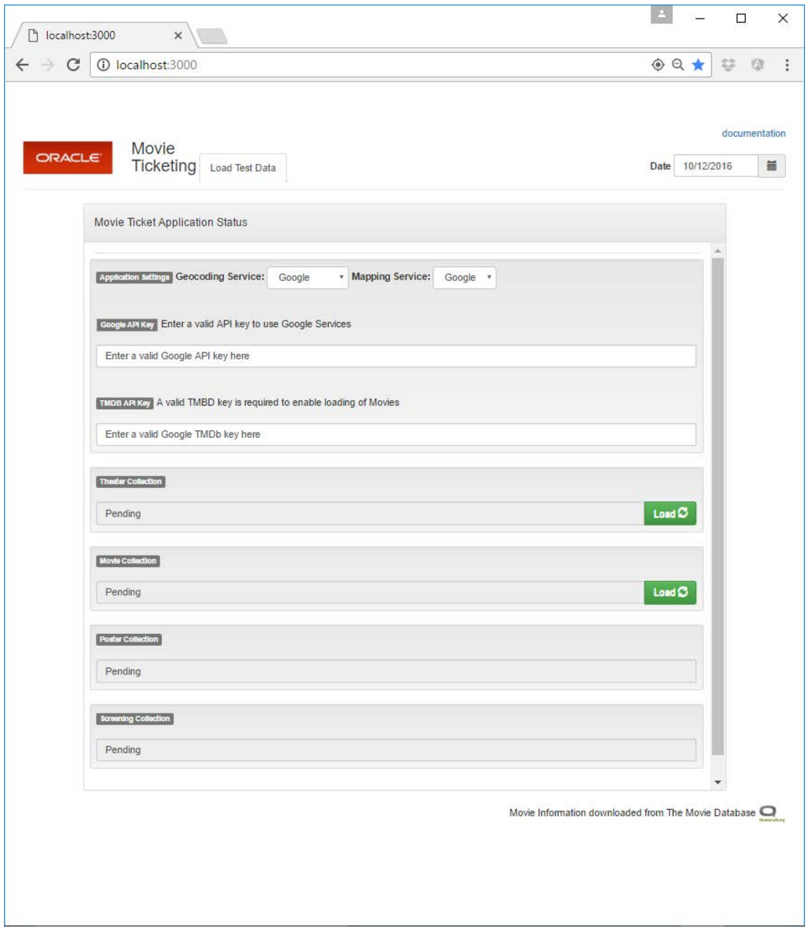
Pending

Movie Information downloaded from The Movie Database

If you want to make use of Geocoding and Mapping please select the required services using the provided menus. In order to load Movie Information a valid key for the TMDb API must be provided using the box marked TMDb API Key. In order to use Google services a valid key for relevant Google APIs must be provided using the box marked Google API. If you have not yet obtained the necessary API keys you can use the 'Get Key' buttons associated with each box to open the appropriate website used to request the necessary keys. Once your selections are made and the required keys have been entered click 'SAVE' to update dataSources.json.

3. Load Theaters and Movies

Successfully saving the configuration will enable the loading of Theaters and Movies.



Movie Ticket Application Status

Application Settings Geocoding Service: Google Mapping Service: Google

Google API Key Enter a valid API key to use Google Services

Enter a valid Google API key here

TMDB API Key A valid TMBD key is required to enable loading of Movies

Enter a valid Google TMDb key here

Theater Collection

Pending Load

Movie Collection

Pending Load

Poster Collection

Pending

Screening Collection

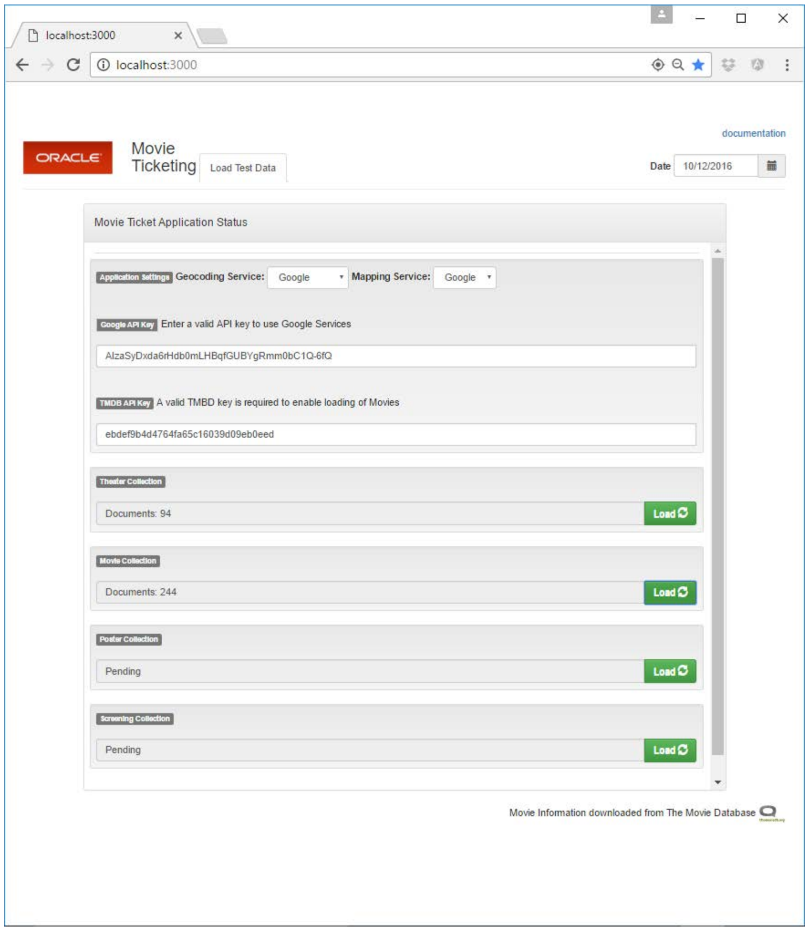
Pending

Movie Information downloaded from The Movie Database

Click the "Load" button associated with the Theaters collection to download theater information from Fandango. Click the "Load" button associated with the Movies collection to download movie information from TMDb. This operation may take a while to complete due to throughput restrictions imposed by the TMDb website. While the load operations are in progress the 'refresh' icon will rotate and the status box associated with target will display 'Working'. Once the load operations are completed the status box will be updated with the number of documents loaded.

4. Load Posters and Screenings

Successfully completing the "Load Movies" operation will enable the loading of Posters. Successfully completing the "Load Movies" and "Load Theaters" operations will enabled the generation of screenings.


localhost:3000

documentation

Movie Ticketing | Load Test Data

Date 10/12/2016

Movie Ticket Application Status

Application Settings Geocoding Service: Google Mapping Service: Google

Google API Key Enter a valid API key to use Google Services

AIzaSyDxda6rHdb0mLHBqfGUBYgRmm0bC1Q-6fQ

TMDB API Key A valid TMBD key is required to enable loading of Movies

ebdef9b4d4764fa65c16039d09eb0eed

Theater Collection

Documents: 94 Load

Movie Collection

Documents: 244 Load

Poster Collection

Pending Load

Screening Collection

Pending Load

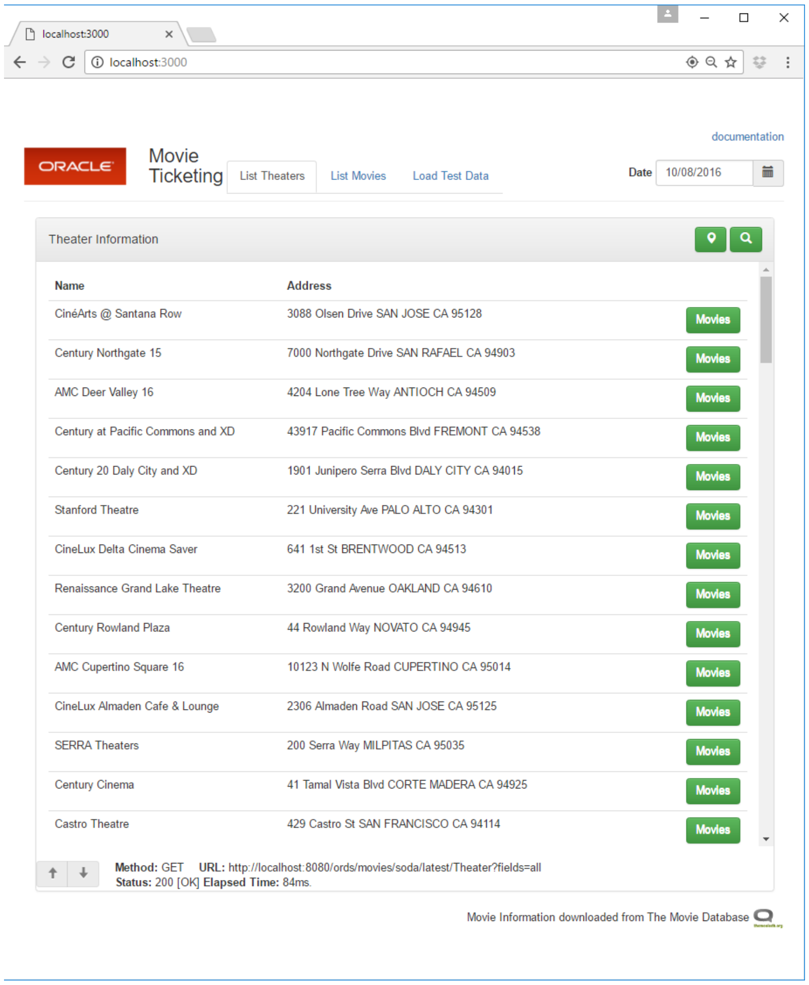
Movie Information downloaded from The Movie Database

Click the "Load" button associated with the Posters collection to download movie posters from TMDb. This operation is also subject to the throughput restrictions imposed by the TMDb website. Click the "Load" button associated with the Screenings collection to generate 2 weeks' worth of screenings from the current set of theaters and movies. While the load and generate operations are is in progress the 'refresh' icon will rotate and the status box associated with target will display 'Working'. Once the load and gemerate operations are completed the status box will be updated with the number of documents loaded.

The data loading process only needs to be completed once, since once data loading is completed all of the information needed to run the application is stored as JSON documents in the Oracle document store. If any of the steps fail, simply click the associated button to try the step again. Note you must successfully complete "Load Movies" before attempting to run "Load Posters", and you must successfully complete "Load Theaters" and "Load Movies" before attempting to run "Generate Screenings". If you re-run either "Load Theaters" or "Load Movies" you must re-run "Generate Screenings before attempting to use the movie ticketing application. If you re-run "Load Movies" you must also re-run "Load Posters" to repopulate the Posters collection.

5. Run the Application

Once the Posters have been loaded and the Screenings generated the application will automatically reload itself and the "List Theaters" and "List Movies" tabs will be displayed. The applicaiton is now ready for use.



Name	Address	
CinéArts @ Santana Row	3088 Olsen Drive SAN JOSE CA 95128	Movies
Century Northgate 15	7000 Northgate Drive SAN RAFAEL CA 94903	Movies
AMC Deer Valley 16	4204 Lone Tree Way ANTIOCH CA 94509	Movies
Century at Pacific Commons and XD	43917 Pacific Commons Blvd FREMONT CA 94538	Movies
Century 20 Daly City and XD	1901 Junipero Serra Blvd DALY CITY CA 94015	Movies
Stanford Theatre	221 University Ave PALO ALTO CA 94301	Movies
CineLux Delta Cinema Saver	641 1st St BRENTWOOD CA 94513	Movies
Renaissance Grand Lake Theatre	3200 Grand Avenue OAKLAND CA 94610	Movies
Century Rowland Plaza	44 Rowland Way NOVATO CA 94945	Movies
AMC Cupertino Square 16	10123 N Wolfe Road CUPERTINO CA 95014	Movies
CineLux Almaden Cafe & Lounge	2306 Almaden Road SAN JOSE CA 95125	Movies
SERRA Theaters	200 Serra Way MILPITAS CA 95035	Movies
Century Cinema	41 Tamal Vista Blvd CORTE MADERA CA 94925	Movies
Castro Theatre	429 Castro St SAN FRANCISCO CA 94114	Movies

Method: GET **URL:** http://localhost:8080/ords/movies/soda/latest/Theater?fields=all
Status: 200 [OK] **Elapsed Time:** 84ms.

Movie Information downloaded from The Movie Database