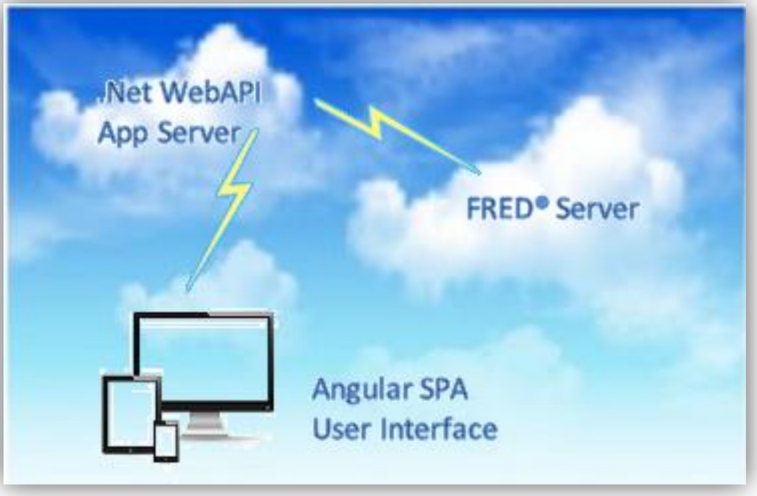
AngularJS Single Page Application (SPA) for FRED® API Toolkit

Overview

This application is an example of how one might consume economic data from the FRED® API via the .Net API toolkit. The application is web-based and provides a browser-oriented user interface based on the Single Page Application (SPA) design paradigm. The UI communicates with a server that leverages the API toolkit to communicate with FRED® API endpoints.

Here is a summary of the application's major components, along with a visual depicting the relationship among the components:

1. The web consumer, implemented as an AngularJS SPA;
2. The server-based service, implemented as a .Net WebAPI assembly, that leverages the API toolkit components and exposes interfaces to their operations;
3. The FRED® API itself, managed by the Federal Reserve Bank of St. Louis.



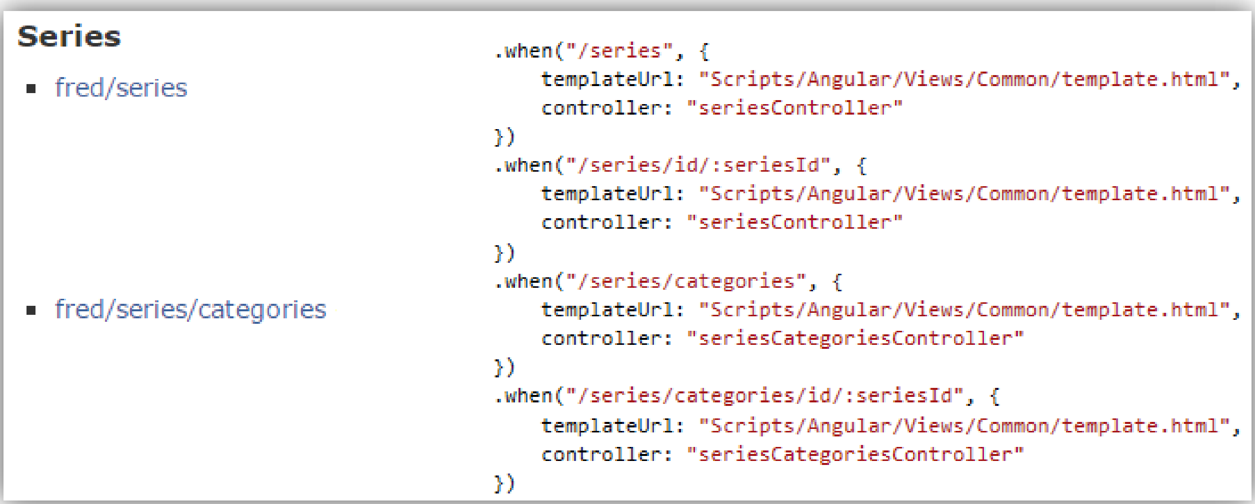
The first two of these components are described in detail below; details on the FRED® API interface can be found at https://research.stlouisfed.org/docs/api/fred.

Web Consumer

The web consumer implements the Model-View-Controller (MVC) design pattern. AngularJS is the MVC framework that provides mediation among MVC components, with requests for data and responses from providers passing through controllers in typical MVC fashion. Each of the three components of the MVC pattern is covered below, following a basic discussion of routes.

Routes

The MVC routes in the web consumer follow FRED® naming conventions as closely as possible, providing routing names that are familiar to users who are accustomed to the FRED® API. Below is an example showing the routes for the first several **Series** API endpoints, including routes that accept RESTful arguments. Notice that a common template is used for each route; this is standard throughout the application.



Series

- fred/series
- fred/series/categories

```
.when("/series", {
    templateUrl: "Scripts/Angular/Views/Common/template.html",
    controller: "seriesController"
})
.when("/series/id/:seriesId", {
    templateUrl: "Scripts/Angular/Views/Common/template.html",
    controller: "seriesController"
})
.when("/series/categories", {
    templateUrl: "Scripts/Angular/Views/Common/template.html",
    controller: "seriesCategoriesController"
})
.when("/series/categories/id/:seriesId", {
    templateUrl: "Scripts/Angular/Views/Common/template.html",
    controller: "seriesCategoriesController"
})
```

Controllers

As the above visual shows, each FRED® API endpoint has its own controller object. These controllers leverage two controller services, one that provides behavior common to all API endpoints, and the other that provides behavior common to a specific FRED® API group: **Categories, Releases**, **Series**, **Sources** and **Tags**. In effect, each API controller provides specialized behavior that complements or overrides the more generic behavior of its API group and of the application as a whole.

The use of composition in the design of the controllers and their services is a result of design refactoring and refinement as much as anything else. The original intent was to use classic

JavaScript prototypal inheritance to reach the goal of the separation, non-proliferation and reusability of code, employing patterns such as Template Method[i]. However, the composition design produced an arguably cleaner result from an implementation standpoint, and one that integrates easier into the Angular framework, so the object-oriented implementation was rejected in favor of composition. On balance, composition, in this application, attains the same goal and is cleaner and easier to understand than its OO counterpart. Perhaps the lesson is: JavaScript is not C#.

The generic services that, together with the specialized controllers, compose the MVC controller layer are stateless, each being a singleton. They expose the following behavior:

	Behavior	Description
Application Base Service		
	initializeApi	Initializes an API specific object for use by the controllers and UI, storing the API object in $scope.
	redirect	Redirects to another page via a route.
	fetch	Requests data from a service and returns the result in $scope.
FRED® API Group Base Service		
	initialize	- Creates an API specific object and delegates to the base controller service (see above) for generic initialization and to the owning controller for specialized initialization. - Provides a base path for redirection by the base controller service (see above). - Synchronizes API group-specific route parameter values with the UI via $scope.
	fetch	Delegates to the base controller service for requests, passing the API group-specific endpoint fragment.

Employing endpoint-specific API objects in the design helps abstract common details of each API into the services that create these objects and away from controllers. This design not only provides encapsulation but also helps minimize the clutter and complexity in the controllers, affording a well-balanced division of concerns. Each controller composition has all the essentials, packaged in a single object, for mediating the interaction between Views and Models.

The API objects, when created, expose the following properties:

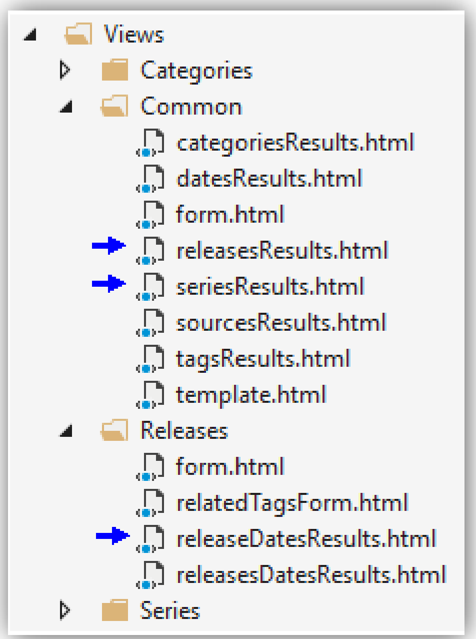
Name	Description
form	The HTML form displayed in the presentation view
hint	An argument hint displayed in the view; this is useful for testing
results	The HTML section in the view that ultimately contains the results of an API fetch
initialOrderByColumn	The name of the column on which data is initially sorted in the view

Controllers can extend the API objects by adding specializations.

Note that, despite the name, the API objects do not contain behavior associated with *accessing* the API-endpoints. This is an MVC design constraint, since the API objects are stored in $scope and available to the Views, which are decoupled from the model.

Views

The web consumer uses a generic template view with boilerplate that is common throughout the application. The visual in the **Routes** section above shows this. Depending on the route the user chooses, Angular imbeds the appropriate sub-views into this template view via the controllers and the configured API objects discussed above. The sub-views are part of the configuration for each API-specific object.


```
Views
  Categories
  Common
    categoriesResults.html
    datesResults.html
    form.html
    releasesResults.html
    seriesResults.html
    sourcesResults.html
    tagsResults.html
    template.html
  Releases
    form.html
    relatedTagsForm.html
    releaseDatesResults.html
    releasesDatesResults.html
  Series
```

There is at least one sub-view, hereafter called simply a “view”, for displaying a form and results for each of the FRED® API groups. These views are stored as a common set, indicating that they are reusable among API operations in each group. The visual at the right shows part of the Views structure with the Common folder expanded.

The choice of a results view begins with the most general view possible and works toward a more specific view as necessary. For example, the /release routes use the generic releasesResults.html file in their views, since that suits those particular cases. The /release/series routes need to display **Series** data, so their views use the generic seriesResults.html file, even though the routes are part of the **Releases** API group. The release/dates routes require an even more specific view; they use the releaseDatesResults.html file. The fact that this file is inside the Releases folder structure indicates that it is specific to the **Releases** API group and not common, i.e., not reusable across API groups.

Form views follow a parallel convention to that used for results views. The visual shows a common form.html file that routes use unless they require a more specialized form. Notice that there is such a specialization within the Releases folder.

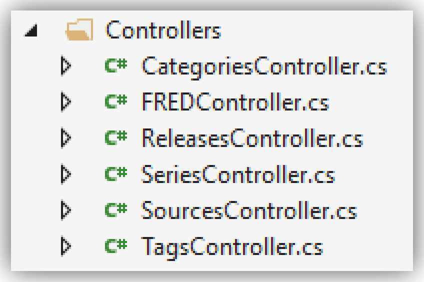
Models

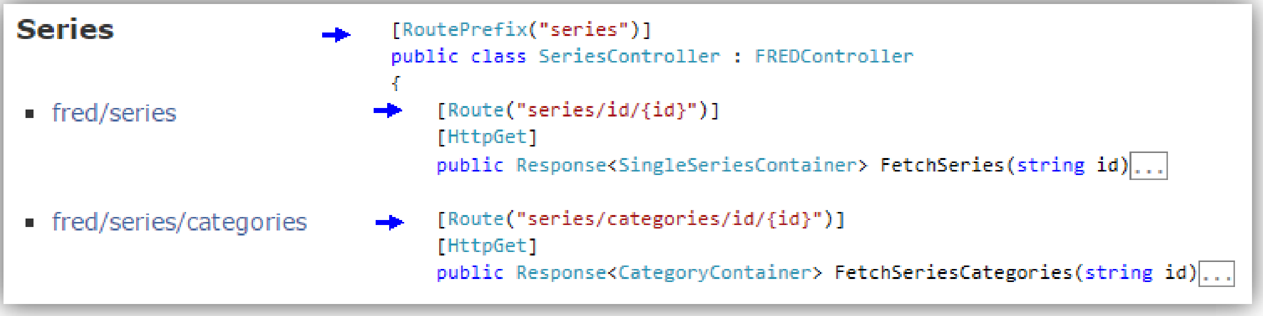
Models for the web consumer application are fairly simple – they are the JSON objects returned from the FRED® API via the solution's WebAPI discussed below. Incidentally, WebAPI is the mechanism used in this solution, but another interface implementation might be used just as well.

WebAPI

The WebAPI provides a server-based interface layer between the web consumer client (the SPA) and the FRED® API. More importantly, the WebAPI leverages the functionality of the .Net API toolkit, exposing properties of the toolkit's classes to the consumer.

The entire interface is segmented by FRED® API group, **Categories, Releases**, **Series**, **Sources** and **Tags**, following the convention used throughout the API toolkit and this sample application. Each group's interface is implemented as an MVC-style controller, deriving from the `FREDController` base class and ultimately from the .Net `ApiController` class. Routing for the WebAPI endpoints follows the same naming conventions as routing for the client consumer, discussed above. Below is an abbreviated example of the methods of the `SeriesController`, showing the routes. These parallel those shown in the visual in the **Routes** section above.





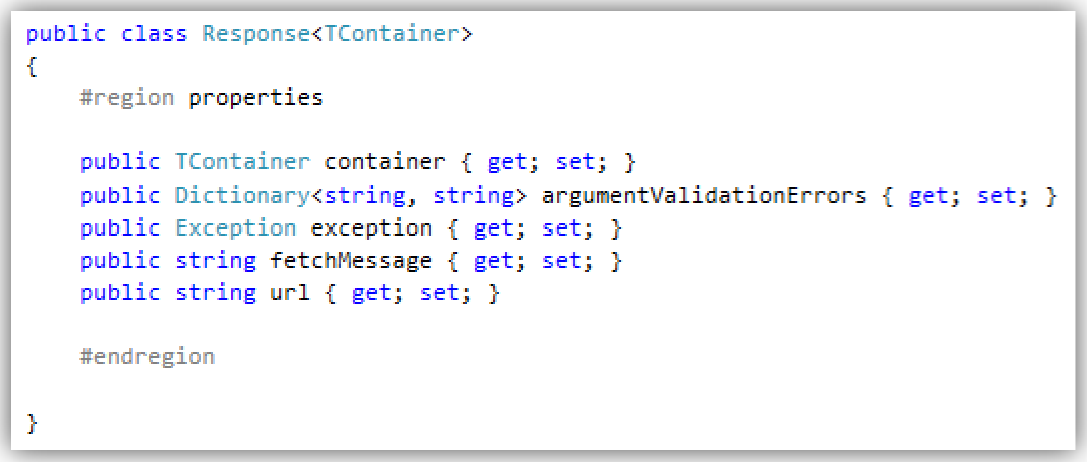
Series

- fred/series
- fred/series/categories

```
[RoutePrefix("series")]
public class SeriesController : FREDController
{
    [Route("series/id/{id}")]
    [HttpGet]
    public Response<SingleSeriesContainer> FetchSeries(string id)...

    [Route("series/categories/id/{id}")]
    [HttpGet]
    public Response<CategoryContainer> FetchSeriesCategories(string id)...
```

The WebAPI exposes each method in its classes as a RESTful endpoint. The response from each method is an object containing the FRED® API response, with its class defined in the argument to the generic Response class, plus additional fields that the API toolkit provides. Below is the structure of the generic Response class.



```
public class Response<TContainer>
{
    #region properties

    public TContainer container { get; set; }
    public Dictionary<string, string> argumentValidationErrors { get; set; }
    public Exception exception { get; set; }
    public string fetchMessage { get; set; }
    public string url { get; set; }

    #endregion

}
```

The .Net framework takes care of serializing the Response into a JSON object that the WebAPI returns to the client.

Appendix A – Document Conventions

These are the typeface and style conventions used throughout this document.

- The typeface and font size used in the body of the document is Calibri 12 point.
- Section names are shown in Cambria 14 point bold.
- References to FRED® API web pages are shown in Cambria 12 point for section names and Verdana 10.5 for active links, for example, **Categories**: fred/category. This is similar to the typeface used on the actual web pages.
- Class names and namespaces are shown in Consolas 11 point, light blue text, for example, `SeriesObservation`.
- Other code is shown in Lucida Sans Unicode 10, for example, category_id.

Notes

---

[i] Design Patterns, Elements of Reusable Object-Oriented Software. Gamma, et al.
Template Method is a Behavioral Pattern. Its description is “Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure.”