You’ve heard of SPA – “single page application” – and wondered what it is. While you could read about it, you’d rather experience it for yourself. But who has time to download a sample? Well if you’ve got Visual Studio, you’ll have a SPA up and running in less than 60 seconds with the “ASP.NET MVC 4 Breeze Single Page Application” template. SHAM-WOW



[[http:/www.breezejs.com/sites/all/images/spa-template/ZephyrRunning.png](http://www.breezejs.com/sites/all/images/spa-template/ZephyrRunning.png)]

# What is Breeze?

# What is the Breeze SPA Template?

Why the template? How to use it. Not to build on directly but to help break through to a larger audience about what SPA is about. Give them that “Aha” moment without which they will not get onboard.

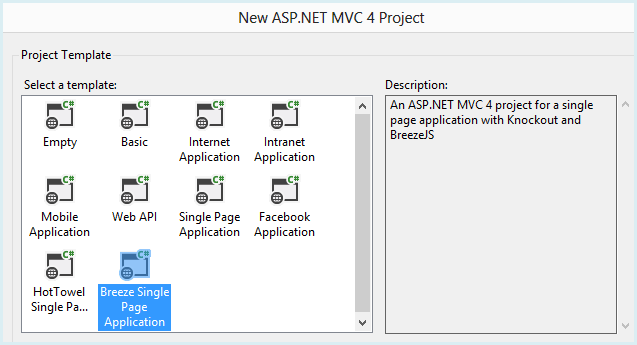
Pros: see John’s Pro/Con. Add: it’s MVC + SPA

[Relationship to the ASP.NET MVC 4 “Single Page Application” template]

Link to [our SPA Template page](http://www.breezejs.com/spa-template?utm_source=ms-spa)

# File | New

[Generate “Zephyr” in the shortest possible steps to Ctrl-F5]



[SelectBreezeSpaTemplate.png]

# Run it

[short tour of Zephyr running .. .maybe a video of it running]

### “Video running” points

Hybrid MVC + SPA … why that’s important

The validation extras

# What’s the point?

Zephyr introduces you to many techniques of end-to-end, Breeze-backed SPA design under the umbrella of ASP.NET MVC

* List them

The glaring omission is screen navigation. The “SP” in SPA refers to the single web page that hosts the client application. The client app itself features fast navigation among multiple screens, governed by JavaScript and HTML in the browser. Such an app asks the server for fresh data, not HTML or workflow.

You’ll want to look elsewhere [#link to notes with links to resources] for examples and guidance on this essential characteristic of JavaScript applications.

But there’s plenty to learn from this simple example. Let’s step inside.

# Solution overview

The Breeze SPA template generates a single-project, web application solution with structure and contents that hew closely to the ASP.NET MVC 4 “Single Page Application” template.

That is entirely intentional. A primary goal of the Breeze template is to demonstrate ***how little you must change*** to add data-rich Breeze development. We’ll concentrate on differences but we can’t do that in a vacuum. So we’ll also cover the commonalities as we go, leaving the deeper details to others [#link to notes with links to resources].

|  |  |
| --- | --- |
|  | **App\_Data** – The *Todo* LocalDb database files; “Show all files” to see them.  **App\_Start** – Server-side web application configuration classes  **Areas** – MVC location for Web API help  **Content** – CSS and theme images  **Controllers** – both MVC page and Web API data controllers  **Filters** – A Web API attribute applied to the AccountController that creates and initializes the database when not found.  **Models** – The server-side, CLR model classes  **Scripts** – The application and 3rd party JavaScript files that drive the client experience  **Views** - HTML for both MVC and SPA views |

[ZephyrSolution.png]

The Zephyr project defines an MVC 4 web application that serves content and data. Most of the 170+ files never leave the server. What’s really interesting about SPA happens on the client. We’ll look at the client first and then circle back to the server

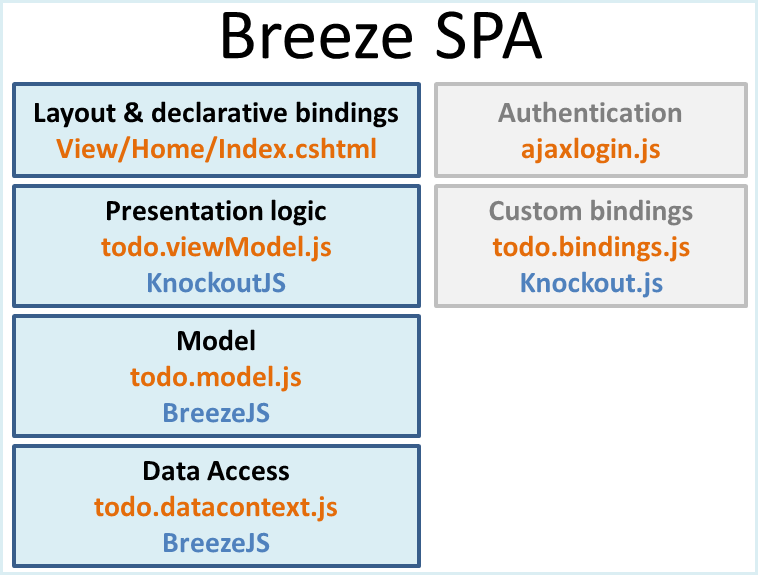
# The Client

In a typical session about 30 file make it over to the browser: 14 style sheets, 15 scripts, and 1 HTML file. We can ignore the style sheets; all but two are themes from jQuery-UI.

Ten of the fifteen scripts are 3rd party libraries:

* [jQuery](http://jquery.com/) (5 libraries and plugins)
* [breezeJS](http://www.breezejs.com/documentation/download) – for data management
* [breeze.savequeuing.js](http://www.breezejs.com/documentation/download) – a Breeze plugin to handle concurrent client save requests.
* [knockoutJS](http://knockoutjs.com/) – for data binding
* [modernizr](http://modernizr.com/) – to reduce the effects of browser differences
* [q.js](https://github.com/kriskowal/q#readme) – a promises library to simplify asynchronous programming, required by Breeze

That leaves an HTML file and the five application scripts located in the ***Scripts/app*** folder.



[ClientArchitecture.png]

We’ll brush past two of the scripts which come to us unadulterated from the ASP.NET SPA template.

|  |  |
| --- | --- |
| **ajaxlogin** | a traditional jQuery JavaScript file in support of the MVC views devoted to authentication tasks: login, logout, and account control. It has no role in the SPA page. |
| **Todo.bindings** | custom Knockout bindings to sweeten the user experience |

The four files in blue define the SPA application proper.

### Views/Home/Index.cshtml

*Index.cshtml* is the SPA shell page. Its main content is divided into pre- and post-authentication views. We won’t cover the pre-authentication views which are well documented for the ASP.NET SPA template.

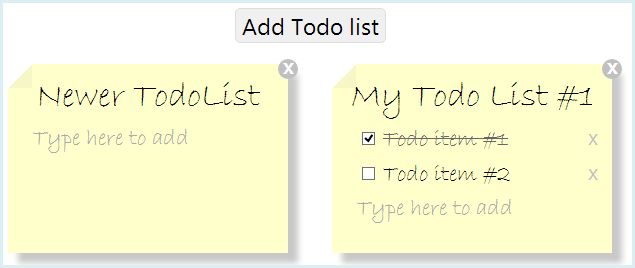
The post-authentication view is on top. It too is almost identical to the original in the ASP.NET SPA template. The only differences:

* An additional click binding to $root.clearErrorMessage in the TodoList *title* binding
* An additional click binding to $root.clearErrorMessage in the TodoItem *title* binding
* The rendering of the Breeze scripts bundle near the bottom of the file.

The ASP.NET SPA template documentation explains how Knockout (KO) binding declarations in this view correspond to properties and methods of the ViewModel (todo.viewmodel.js). We’ll briefly recap.

KO is a data binding library in MVVM (Model-View-ViewModel) style. You decorate HTML elements in the View with special KO binding attributes. Those attributes identify corresponding properties or methods in a JavaScript object called the ViewModel. When the user enters values or clicks a button, KO pushes values into the ViewModel properties or invokes methods. If the ViewModel properties are “observable”, KO can detect and push changes to those properties back into the View HTML.

The app presents one or more TodoLists in the guise of sticky notes.



[TwoTodoLists.png]

The outer markup for the TodoLists follows

<p class="error" data-bind="text: error"></p>

<button data-bind="click: addTodoList">Add Todo list</button>

<section id="lists" data-bind="foreach: todoLists, visible: todoLists().length > 0">

<article class="todoList"> ... </article>

</section>

The <p> element shows the most basic Knockout (KO) binding. The “**data-bind**” attribute is the KO binding. It binds the paragraph text to the contents of the ViewModel’s “error” property. The ViewModel, in this case, is an object created by the *todo.viewmodel.js* script which we’ll look at soon.

The click action of the <button> element is bound to the addTodoList method of the ViewModel. We hope you’re starting to get the picture.

#### Knockout repeaters and templates

The <section> element is bound with a KO repeater instruction. In plain English, it says

“Take each item from the ViewModel’s todoLists array property and bind it to the template defined in the <article> element.”

The “visible: ...” binding tells KO to reveal the repeater only if there is at least one TodoList.

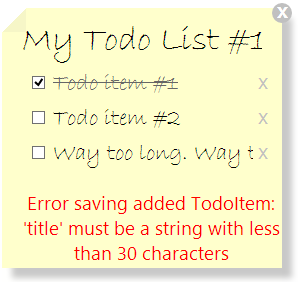
Rummage around in the <article> template and you’ll find another repeater.

<ul data-bind="foreach: todos">

<li>... </li>

</ul>

Now the TodoList is serving as the ViewModel. The TodoList has a todos property with zero or more TodoItems. This repeater creates an <li> element for each TodoItem and binds it to the template defined within the <li>. Here are three TodoItems bound to this template followed by the template itself:



<li>

<input type="checkbox" data-bind="checked: isDone" />

<input class="todoItemInput" type="text"

data-bind="value: title,

disable: isDone,

blurOnEnter: true,

click: $root.clearErrorMessage" />

<a href="#" data-bind="click: $parent.deleteTodo">X</a>

<p class="error" data-bind="visible: errorMessage, text: errorMessage"></p>

</li>

Some observations:

* The *disable* binding add the “disable” attribute to the <input> element when the TodoItem is done, thus preventing changes to its *title*.
* A CSS selector in *TodoList.css* adds the strikethrough and lightens the *title* text.
* KO updates a ViewModel property when the element loses focus.
* *blurOnEnter* listens for the Enter key; when heard, it forces “lost focus” which causes KO to update the *title* property. This is one of the custom KO bindings defined in *todo.bindings.js.*
* clicking in clears the error message … as we’ll explain in a moment
* The little “x” after the title is a link bound to the deleteToDo method. This method is not on the TodoItem but on its parent TodoList. We’ll be looking for that property when we examine the TodoList initializer in *todo.model.js*.
* A TodoItem has an errorMessage property which, if other than an empty string, will appear in a paragraph below the title as we see in the third example TodoItem.

This is the same HTML and the same bindings we’d see in an app generated from the ASP.NET SPA template … except for the click binding to $root.clearErrorMessage.

The “$root” token tells KO to look for a method at the root of the binding tree, the clearErrorMessage method of the ViewModel defined in *todo.viewmodel.js*. We added this feature to the Breeze sample so the user could clear away the error message while fixing the title; the persistence of the error message had been a usability annoyance.

### ViewModel

The *todo.viewmodel.js* file defines the app’s lone ViewModel. This ViewModel exposes five members for binding to the View. Examine the KO data bindings and you’ll see that they refer either to one of these five members or to a member of a Todo entity. This is the recommended strategy for your Knockout MVVM applications.

The five data bound members belong to the ViewModel object returned at the bottom of the module.

return {

todoLists: todoLists, // observable array of TodoList objects

error: error, // application error message

addTodoList: addTodoList, // creates and saves a new TodoList

deleteTodoList: deleteTodoList, // delete a TodoList and its child TodoItems

clearErrorMessage: clearErrorMessage // clear an object’s error message

};

This is a good example of the Revealing Module Pattern described by [Dan Wahlin in this PluralSight post](http://blog.pluralsight.com/2012/10/02/revealing-module-pattern-structuring-javascript-code-part-iii/).

The last line of the file binds the View to the ViewModel bringing application life to the screen.

ko.applyBindings(window.todoApp.todoListViewModel);

### ViewModel invariance

One of the design goals for the Breeze template was to show that **a properly designed View and ViewModel can be independent of the supporting data modeling and management layers**.

For this reason, the Breeze SPA template View and ViewModel are virtually identical to their counterparts in the ASP.NET SPA template. The only substantive difference was the necessary inclusion of Breeze library scripts in the **Index.cshtml**.

As a bonus we added the error message clearing feature. The extra clearErrorMessage method adds four lines to what would otherwise be an identical todo.viewmodel.js file. We could back-port this to the ASP.NET SPA template “as is”.

The two template applications have very different model and data access implementations. Their ViewModels can be identical because they stick to their responsibility: **support the View**. They delegate their model and data access differences to a separate datacontext component.

### DataContext

The *todo.datacontext.js* file defines the module that handles all remote data access.

# The Server

The web application project structure should be familiar to anyone acquainted with MVC 4 projects. In this section we’ll highlight a few of the server-side files that are specific to the Breeze version of the SPA template application.

#### App\_Start/BreezeWebApiConfig

The App\_Start folder holds web application configuration detail classes. The Breeze SPA template added the *BreezeWebApiConfig* file to **define a Breeze Web API route** and **position it in front** of the default Web API routes. Here’s the route:

GlobalConfiguration.Configuration.Routes.MapHttpRoute(

name: "BreezeApi",

routeTemplate: "api/{controller}/{action}"

);

The typical Breeze application sends data requests to a single Web API controller. The URL identifies the controller and an action to perform. Here’s a request for all TodoLists:

http://localhost:60124/api/Todo/TodoLists

We’ll look at this again soon when we examine the controller. For now we observe that the “Todo” path segment maps to the {controller} token and “TodoLists” to the {action} token.

This routing scheme differs from the default Web API route defined in WebApiConfig. Rather than modify that file we added BreezeWebApiConfig and gave its route precedence via the WebActivator assembly attribute at the top of the file:

[assembly: WebActivator.PreApplicationStartMethod(...)

### App\_Start/BundleConfig

This application uses [Web Optimization](http://www.asp.net/mvc/tutorials/mvc-4/bundling-and-minification) to bundle and minify files delivered to the browser. We’re using Breeze so we’ve added a bundle of Breeze-related scripts:

bundles.Add(new ScriptBundle("~/bundles/breeze").Include(

"~/Scripts/q.js",

"~/Scripts/breeze.debug.js",

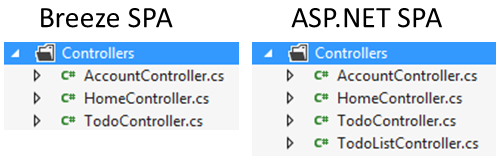
"~/Scripts/breeze.min.js",

"~/Scripts/breeze.savequeuing.js"));

* [q.js](https://github.com/kriskowal/q#readme) – a promises library to simplify asynchronous programming
* [breeze.debug.js](http://www.breezejs.com/documentation/download) – the un-minified breeze library
* [breeze.min.js](http://www.breezejs.com/documentation/download) – the minified breeze library. Web Optimization will only bundle one or the other of the Breeze libraries depending upon whether you build the application for debug or release.
* [breeze.savequeuing.js](http://www.breezejs.com/documentation/download) – a Breeze plugin to handle concurrent client save requests.

### Controllers

By convention, the *Controllers* folder holds both MVC and Web API controllers.



The two MVC controllers, *HomeController* and *AccountController*, are the same in both templates. The *HomeController* serves the SPA host page; the *AccountController* handles authentication-related requests (registration, login, logout, etc.).

The ASP.NET SPA app follows the “controller-per-model-type” resource-oriented approach. This has a two-entity model so the ASP.NET SPA has two controllers. A Breeze app follows the action-oriented approach and typically only needs Web API controller to cover requests for the entire model.

It only needs one Web API controller because the number of requests it must handle is comparatively few.

### TodoController

The TodoController exposes only three methods:

|  |  |
| --- | --- |
| **Metadata** | Get metadata about the server model |
| **SaveChanges** | Save a bundle of entity changes |
| **TodoLists** | Get “Todolist” data, potentially filtered, ordered, paged, and extended with child Todos |

#### Metadata and SaveChanges

The first two are standard fare. A Breeze client needs metadata describing the model. Most Breeze developers prefer to get metadata from the server and this controller can supply the metadata.

The ASP.NET SPA uses jQuery AJAX calls to save each change one item at a time. Look at one of its controllers and you’ll see PUT, POST, and DELETE methods to update, add, and delete single items.

A Breeze app saves multiple changes as a bundle in a single transaction. We could create a new TodoList, add new TodoItems, modify another TodoList, and delete a third (along with its child Todos). We could POST all of these changes one change-set to this controller’s SaveChanges method. This particular sample isn’t written to do that … it would have required a much more complicated ASP.NET SPA sample. The Breeze version might have been less complicated if we had.

#### Query methods

The ASP.NET sample has a single GET method, TodoListController.GetTodoLists. The Breeze version has a single GET method, TodoController.Todos. Both return TodoLists with their associated child TodoItems.

Here’s the ASP.NET SPA original:

public IEnumerable<TodoListDto> GetTodoLists() {

return db.TodoLists.Include("Todos")

.Where(u => u.UserId == User.Identity.Name)

.OrderByDescending(u => u.TodoListId)

.AsEnumerable()

.Select(todoList => new TodoListDto(todoList));

}

It filters for the current user’s TodoLists. Then it sorts by the key in descending order. Then it casts to Enumerable so it can pass data to a DTO class which fetches the related child TodoItems.

The Breeze TodoController.Todos is much simpler:

[HttpGet]

public IQueryable<TodoList> TodoLists() {

return \_repository.TodoLists;

}

Sure it delegates to a repository – that’s where we moved the filter for the current user’s TodoLists. We’ll get to the repository shortly. The key point is that it returns an IQueryable, leaving the matter of sort order and inclusion of child TodoItems to the client’s discretion.

The client can filter, page, order, expand, and select a subset of TodoList properties by sending OData queries to this action:

http://localhost:60124/api/todo/todolists?$filter=TodoListId eq 1

http://localhost:60124/api/todo/todolists?$orderby=Title

http://localhost:60124/api/Todo/TodoLists?$orderby=Title&$skip=1&$top=1

http://localhost:60124/api/Todo/TodoLists?$expand=Todos

The client may be able to satisfy all of its data retrieval requirements by this single query method alone, thus maintaining a lean controller. When a single IQueryable method won’t suffice, you can add specialized GET actions to the controller and call them from the Breeze client.

It follows that the controller grows slowly, as slowly as one query method per exposed model type. For example, if the client needed TodoItems, independent of their parent TodoLists, we could add a fourth “*TodoItems*” query action method similar to the TodoLists method.

Aside from granting the client more flexibility for free, this approach also performs much better. The client-specified filtering, ordering, paging, and expand will execute on the data tier, not on the server tier.

The original ASP.NET SPA query casts to IEnumerable which means the server will fetch every one of the user’s TodoLists even if the client queried for only one of them. Worse, the DTO makes a second trip to the database to get the child TodoItems for each and every TodoList.

Of course you’ll never notice in this application which doesn’t do any serious querying. Upon launch it asks for all TodoLists belonging to the current user and that’s the last time it asks for any data. We’ll have to imagine how the controllers will be affected by richer client requirements.

#### TodoController attributes

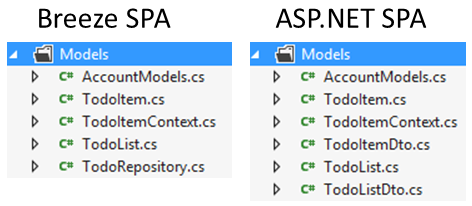
The Breeze controller is decorated with two attributes

|  |  |
| --- | --- |
| **Authorize** | The Web API attribute that prevents unauthenticated users from reaching the controller. |
| **BreezeController** | Configures this controller to talk to Breeze clients. It replaces all filters with a [Newtonsoft Json.NET](http://james.newtonking.com/pages/json-net.aspx) filter configured for Breeze that serializes and deserialized data as JSON. It also installs the Breeze Action filter that interprets OData query parameters and applies them to the IQueryable objects returned by such GET action methods as TodoLists; it replaces a similar Web API filter for this controller only. |

### Models

The SPA sample apps rely on Entity Framework (EF) Code First models. The model classes and the DbContext are almost identical.

The only difference: we added **MaxLength** attributes to the **Title** properties so we could demonstrate Breeze validation. Breeze includes the maximum string length and required constraints in the metadata it sends to the client. We get these validations for free in Breeze. The user learns about validation errors immediately, without a delayed rejection from the server. Of course EF will also apply these validations on the server automatically; that’s a feature of EF’s DbContext.



### Models/TodoRepository

The three methods of the TodoController delegate to a TodoRepository class. We’re conforming to a general principle of Web API controller design which argues for simple implementations. The controller is supposed to direct traffic, not do heavy work of its own.

The TodoRepository is in the *Models* folder. Its primary purpose is to mediate between the Web API controller and the Entity Framework which provides access to the database (as it does for the ASP.NET SPA app).

The TodoRepository inherits from a Breeze.NET helper class, the EFContextProvider, which vastly simplifies interactions with the Entity Framework. You can read about it [here](http://www.breezejs.com/documentation/custom-efcontextprovider).

The repository is a good place to start putting server-side business logic. A typical demo wouldn’t have any which is why many samples simply delegate directly to the EFContextProvider.SaveChanges method. This SPA sample actually has some business logic. Let’s take a peek.

#### Query constraints

Here is the TodoLists query to which the TodoController delegates its own TodoLists GET method:

public DbQuery<TodoList> TodoLists {

get {

return (DbQuery<TodoList>)Context.TodoLists

.Where(t => t.UserId == UserId);

}

}

Context is a typical Entity Framework DbContext object sporting named DbQuery properties. In this app a user may only see her own TodoLists; we’re enforcing that rule in the repository rather than in the controller.

#### Save constraints

The server shouldn’t trust the client. That’s especially important with save requests. You should validate client data for data integrity and authorization before storing them to the database.

The JSON payload of a Breeze save request is deserialized by [Newtonsoft Json.NET](http://james.newtonking.com/pages/json-net.aspx) into a “bundle” object representing an entity change-set. The EFContextProvider.SaveChanges method (a) dissects the bundle, (b) prepares the Entity Framework (EF) Context for the save, and (c) calls Context.SaveChanges on your behalf.

Between (a) and (b) it gives you two chances to intervene. You can intercept each entity before the provider adds it to the Context by overriding BeforeSaveEntity. In your override you can approve the entity (return true), exclude the entity silently (return false), or throw an exception that terminates the entire save.

You don’t want too much logic in the repository so, in a bigger app, you probably would treat this override as a dispatcher to type-specific validation classes. But this is a small sample so we validate the TodoItem and TodoList entities right here in the repository.

We validate a TodoList as follows:

private bool BeforeSaveTodoList(TodoList todoList, EntityInfo info)

{

if (info.EntityState == EntityState.Added)

{

todoList.UserId = UserId;

return true;

}

return UserId == todoList.UserId || throwCannotSaveEntityForThisUser();

}

If the TodoList is new, we assign it to the current user. Otherwise (modify and delete), we confirm that the TodoList belongs to the current user and throw an exception it if isn’t. We’re imposing the same logic you’ll find in the PUT, POST, and DELETE methods of the ASP.NET SPA sample’s TodoListController. Compare for yourself.

Your second opportunity to intervene is the BeforeSaveEntities virtual method which gives you a chance to inspect the entire change-set as a whole. We don’t use that feature in this sample.

## What is Breeze.NET?

You thought Breeze was a JavaScript library. Yet we’re still on the server describing Breeze components such as the EFContextProvider.

BreezeJS is a pure JavaScript technology. You do not need any Breeze on the server. The server need not use use any Microsoft technology: not IIS, not Web API, not Entity Framework, not SQL server.

But this SPA sample is generated from an ASP.NET MVC template. That means .NET on the backend. We’ve made it easier to program a .NET backend with Breeze.NET components in support of the BreezeJS pure JavaScript technology on the client. A casual comparison with the hand-rolled save logic in the ASP.NET SPA sample reveals just how helpful Breeze.NET can be.