VOLT

Table of Contents

Introduction

Volt is a Ruby web framework where your Ruby code runs on both the server and the client (via opal). The DOM automatically updates as a user interacts with the page. Page state can be stored in the URL. If the user hits a URL directly, the HTML will first be rendered on the server for faster load times and easier indexing by search engines, then be maintained in the client.

Instead of syncing data between the client and the server via HTTP, Volt uses a persistent connection. When data is updated on one client, it is updated in the database and in any other listening clients (with almost no setup needed).

Page HTML is written in a template language where you can put Ruby between `{{` and `}}`. Volt uses data flow/reactive programming to automatically and intelligently propagate changes to the DOM (or any other code that wants to know when a value changes). When something in the DOM changes, Volt intelligently updates only the nodes that need to be changed.

See some demo videos here:

- Volt Todos Example
- Pagination Example
- Build a Blog with Volt ** Note: The blog video is outdated, expect an updated version soon.

Check out demo apps:

- https://github.com/voltrb/todos3
- https://github.com/voltrb/contactsdemo

Goals

Volt has the following goals:

1. Developer happiness
2. Sharing of code between the client and server
3. Automatic data syncing between client and server
4. Apps are built as nested components. Components can be shared (via gems)
5. Concurrent. Volt provides tools to simplify concurrency. Component rendering is done in parallel on the server
6. Intelligent asset management
7. Secure (shouldn't need to be said, but it does)
8. Fast, light, and scalable
9. Understandable code base
10. Upgradeability

Road Map

Many of the core Volt features are implemented. We still have a bit to go before 1.0, most of it involving models.

1. Model read/write permissions
2. User accounts, user collection, signup/login templates

Tutorial

This guide will take you through creating a basic web application in Volt. This tutorial assumes a basic knowledge of Ruby and web development. Also this tutorial is a work in progress :-)

Setup

First, lets install Volt and create an empty app. Be sure you have ruby (>2.1.0) and rubygems installed.

Next install the volt gem:

```
gem install volt
```

Using the volt gem, we can create a new project:

```
volt new sample_project
```

This will setup a basic project in the sample_project folder. We can `cd` into the folder and run the server:

```
bundle exec volt server
```

Lastly, we can access the Volt console with:

```
bundle exec volt console
```

A Sample Todo App

Sometimes the easiest way to learn a new piece of technology is to start building. Our first example project will be a simple todo app like the ones on todomvc.com. Our app will have many simple features:

- A field where users type in a todo and press enter to add it
- A list of todos
    - A checkbox to complete the todo
    - An X button to remove the todo
- A display of the number of remaining todos
- A button to complete all tasks (that is disabled if all tasks are already complete)

If you want, you can view this tutorial in video form and follow along.

To start, let's generate a new app:

```
gem install volt
volt new todo_example
cd todo_example
```

You'll notice that Volt created a `todo_example` folder and filled it with the scaffolding for a new Volt project, along with other common things like a Gemfile and sensible .gitignore. Volt apps are built as nested components, and your app starts with a component called `main`, which has a controller and some views.

To run the template server:

```
bundle exec volt server
```

When you save changes to a file, volt will automatically reload the file and push the changes to anyone viewing your page. Let's create a new page while leaving the server running.

First, create a new file, `app/main/views/main/todos.html` and give it some basic content:

```
<:Title>
  Todos

<:Body>
  <h1>Todo List</h1>
```

And then add a `/todos` link to the navbar, which is rendered from `app/main/views/main/main.html`:

```
...
<:Body>
  <div class="container">
    <div class="header">
      <ul class="nav nav-pills pull-right">
        <:nav href="/" text="Home" />
        <:nav href="/todos" text="Todos" /> <!-- New link -->
        <:nav href="/about" text="About" />
      </ul>
...
```

And also add a route for todos in `app/main/config/routes.rb`:

```
get '/about', _action: 'about'
get '/todos', _action: 'todos' # New route
...
```

Once all these changes are saved, you will be able to navigate to the page we created for the Todo List.

Next, we want to add a way for users to add a todo to the list with a form, so we'll start by adding to the body of `todos.html`:

```
...
<:Body>
  <h1>Todo List</h1>

  <form e-submit="add_todo" role="form">
    <div class="form-group">
      <label>Todo</label>
      <input class="form-control" type="text" value="{{ page._new_todo  }}" />
    </div>
  </form>
```

Anything in `{{ }}` is executed as Ruby code, both on the client and the server, so we are binding the value of the form to a member of the `page` collection. In Volt, there are a number of different collections, `page` is just a temporary collection, and will be lost if you navigate away or refresh. Any value that gets bound in the view will be automatically updated in all places, so `page._new_todo` is accessible from other parts of your code. We'll take advantage of this by adding a method to `app/main/controllers/main_controller.rb`:

```
...
def add_todo
  page._todos << { name: page._new_todo }
  page._new_todo = ''
end
...
```

This method will append a hash to `page._todos` with the value of `page._new_todo` and clear out `page._new_todo`. To be able to fully appreciate the `page._todos` collection, we'll add a table to our page:

```
...
<:Body>
  <h1>Todo List</h1>

  <table class="todo-table">
    {{ page._todos.each do |todo| }}
      <tr>
        <td>{{ todo._name }}</td>
      </tr>
    {{ end }}
  </table>
...
```

Now, once everything is saved and reloads, any time you submit by hitting enter, it'll add to the list and clear out the form. Volt is reactive and intelligent, so any time the list is updated, only the new elements will be drawn; it won't redraw the entire list.

More Todo Functionality

Sure, now we have a list, but things don't start to get interesting until you build additional functionality on top of that list.

To be able to remove items from the todo list, we'll need a method in the controller as well as a button linked to that method. We'll define a `remove_todo` method in `app/main/controllers/main_controller.rb`:

```
...
def remove_todo(todo)
  page._todos.delete(todo)
end
...
```

And in `app/main/views/main/todos.html`, add a button to the table of todos:

```
...
<tr>
  <td>{{ todo._name }}</td>
  <td><button e-click="remove_todo(todo)">X</button></td>
</tr>
...
```

Obviously, our todo list also needs to be able to monitor which items have been completed. If we simply added a checkbox it wouldn't be too interesting, but because things are synced everywhere we can use that value in a number of ways. We're going to apply some CSS to completed items in the list.

```
...
<tr>
  <td><input type="checkbox" checked="{{ todo._completed }}" /></td>
  <td class="{{ if todo._completed }}complete{{ end }}">{{ todo._name }}</td>
  <td><button e-click="remove_todo(todo)">X</button></td>
</tr>
...
```

Here's some CSS that we'll use to make the todos prettier. In Volt, all CSS and JavaScript is included by default, so you rarely have to mess with require tags or script tags. You can just drop this into `app/main/assets/css/app.css.scss`:

```
textarea {
  height: 140px;
  width: 100%;
}

.todo-table {
  width: auto;

  tr {
    &.selected td {
      background-color: #428bca;
      color: #FFFFFF;

      button {
        color: #000000;
      }
    }

    td {
      padding: 5px;
      border-top: 1px solid #EEEEEE;

      &.complete {
        text-decoration: line-through;
        color: #CCCCCC;
      }
    }
  }
}
```

Now you can see that checking and unchecking things updates the state right away.

Another feature we might want to add is the ability to select a todo and add an extra description to it. At this point we will also add a few grid framework (Bootstrap) placement classes to make the layout look a little nicer. We'll do this by adding more to our view:

```
...
<:Body>
  <div class="row">
    <div class="col-md-4">

      <h1>Todo List</h1>

      <table class="todo-table">
        {{page._todos.each do |todo| }}
        <!-- Use params to store the current index -->
        <tr
          e-click="params._index = index"
          class="{{ if params._index.or(0).to_i == index }}selected{{ end }}"
          >
          <td><input type="checkbox" checked="{{ todo._completed }}" /></td>
          <td class="{{ if todo._completed }}complete{{ end }}">{{ todo._name }}</td>
          <td><button e-click="remove_todo(todo)">X</button></td>
        </tr>
        {{ end }}
      </table>

      <form e-submit="add_todo" role="form">
        <div class="form-group">
          <label>Todo</label>
          <input class="form-control" type="text" value="{{ page._new_todo }}" />
        </div>
      </form>
    </div>

    <!-- Display extra info -->
    <div class="col-md-8">
      {{ if current_todo }}
      <h2>{{ current_todo._name }}</h2>

      <textarea>{{ current_todo._description }}</textarea>
      {{ end }}
    </div>
  </div>
...
```

The new stuff in our table makes it so that any time you click on a todo it sets the index in Volt's params collection, which equates to the URL params which are automatically updated. Because values in the params collection could be unassigned, we use `#or` to provide a default value, and then apply some extra CSS to the selected todo.

The new section at the bottom says that if there is a `current_todo`, we want to display some extra details about it. For this to work, we're going to need another method in our controller:

```
...
def current_todo
  page._todos[params._index.or(0).to_i]
end
...
```

Now, when you click on a todo, the notes section of the page will update automatically.

Controllers in Volt inherit from `ModelController` which means you can assign a model to your controller and any missing methods on the controller get passed to the model. We're going to back up all of our todos to a database by importing a model:

```
class MainController < Volt::ModelController
  model :store

...
```

Now we can replace all references to `page._todos` with `_todos` (in both the controller and the view) and our todos will persist to the database. We just need to have Mongo running.

If you've never used Mongo before, you can find instructions for installing it on your operating system on their website, under Installation Guides. As mentioned in their instructions, be sure that the user who is going to be running Mongo has read and write permissions for the `/data/db` directory. If you'd like to run Mongo as the user that you are currently logged in as without using `sudo` or similar, be sure to run `sudo chown $USER /data/db` after you've created the directory.

Once you have Mongo installed, you can start it as either a background process, or by simply running `mongod` in a separate terminal. As long as it's running, and you are using `_todos` in the view and controller, Volt will now automatically sync these values to any open clients.

Go ahead and try opening your Todos page in a few different windows and make some changes to one of the Todo items. If you've set up Mongo correctly, you should see your changes get pushed from the server to each of the clients.

From here it's simple to add a couple more features to our list:

```
...
<:Body>
  <div class = "row">
    <div class = "col-md-4">

      <h1>{{ _todos.size }} Todo List</h1>
...
```

This will show the number of current todo items and will update automatically.

If we want to manage multiple todos at once, we can take advantage of the fact that Volt collections support the methods of normal Ruby collections.

```
...
def check_all
  _todos.each { |todo| todo._completed = true }
end

def completed
  _todos.count { |t| t._completed }
end

def incomplete
  _todos.size - completed
end

def percent_complete
  (completed / _todos.size.to_f * 100).round
end
...
```

Now we can add a button that checks all items at once and a progress bar which will show how many of the items we've done so far.

```
...
<h1>{{ _todos.size }} Todo List</h1>

<button e-click="check_all">Check All ({{ incomplete }})</button>

<div class="progress">
  <div class="progress-bar" role="progressbar" style="width: {{ percent_complete }}%;" >
```

```
      {{ percent_complete }}%
    </div>
  </div>
  ...
```

As you can see, Volt makes it possible to do interactive things like this without writing a lot of code, and the syncing properties make it really natural to create realtime web apps.

Volt Documentation

The following attempts to document all of the features of Volt. While the code is always the final source of authority, we attempt to keep these docs as up-to-date as possible.

Rendering

When a user interacts with a web page, typically we want to do two things:

1. Change application state
2. Update the DOM

For example, when a user clicks to add a new todo item to a todo list, we might create an object to represent the todo item, then add an item to the list's DOM. A lot of work needs to be done to make sure that the object and the DOM always stay in sync.

The idea of "reactive programming" can be used to simplify maintaining the DOM. Instead of having event handlers that manage a model and manage the DOM, we have event handlers that update reactive data models. In this style of programming, we describe our DOM layer in a declarative way and it automatically knows how to render and stay up-to-date with what's in our our data models.

State and Computations

Web applications center around maintaining state. Many events can trigger changes to a state. Page interactions like entering text into form elements, clicking a button or link, scrolling, etc. can all change the state of the app. In the past, each page interaction event would manually change any state stored on a page.

In Volt, to simplify managing application state, all application state is kept in models that can optionally be persisted in different locations. By centralizing the application state, we reduce the amount of complex code needed to update a page. We can then build our page's html declaratively. The relationships between the page and it's models are bound using function and method calls.

We want our DOM to automatically update when our model data changes. To make this happen, Volt lets you "watch" any method/proc for updates.

Computations

Lets take a look at this in practice. We'll use the `page` collection as an example. (You'll learn more about collections later.)

First, we setup a computation watch. Computations are built by calling `.watch!` on a Proc. Here we'll use the ruby 1.9 proc shorthand syntax `-> { ... }`. It will run once, then run again each time the data in `page._name` changes.

```
page._name = 'Ryan'
-> { puts page._name }.watch!
# => Ryan
page._name = 'Jimmy'
# => Jimmy
```

Each time `page._name` is assigned to a new value, the computation is run again. A re-run of the computation will be triggered when any data accessed in the previous run is changed. This lets us access data through methods and still have watches be re-triggered.

```
page._first = 'Ryan'
page._last = 'Stout'

def lookup_name
  return "#{page._first} #{page._last}"
end

-> do
  puts lookup_name
```

```
end.watch!
# => Ryan Stout

page._first = 'Jimmy'
# => Jimmy Stout

page._last = 'Jones'
# => Jimmy Jones
```

When you call `.watch!`, the return value is a `Volt::Computation` object. In the event that you no longer want to receive updates, you can call `.stop` on the computation.

```
page._name = 'Ryan'

comp = -> { puts page._name }.watch!
# => Ryan

page._name = 'Jimmy'
# => Jimmy

comp.stop

page._name = 'Jo'
# (nothing)
```

Dependencies

TODO: Explain Dependencies

As a Volt user, you rarely need to use Computations and Dependencies directly. Instead, you usually just interact with models and bindings. Computations are used under the hood, and having a full understanding of what's going on is useful, but not required.

Views

Views in Volt use a templating language similar to handlebars. They can be broken up into sections. A section header looks like the following:

```
<:Body>
```

Section headers should start with a capital letter so as not to be confused with controls. Section headers do not use closing tags. If no section header is provided, the `:Body` section is assumed.

Sections help you split up different parts of the same content (title and body usually), but within the same file.

Bindings

In Volt, views are written in a simple template language where ruby can be inserted anywhere between `{{` and `}}`. Volt lets you use the usual flow control statements in views (`if` , `elsif` , `else` , and `each`). You can also render other views using the `template` binding.

Controller Backing

While we use the controller terminology, Volt is closer to a MVVM framework. Any method call or instance variable lookup runs in the context of a controller.

If you have a view at `app/home/views/index/index.html` it will load the controller at `app/home/controller/index_controller.rb` .

Content binding

The most basic binding is a content binding:

```
<p>Hello {{ name }}</p>
```

The content binding runs the Ruby code between `{{` and `}}`, then renders the return value. Any time the data a content binding relies on changes, the binding will run again and update the text. Text in content bindings is html escaped by default.

If binding

An `if` binding lets you provide basic flow control.

```
{{ if _some_check? }}
  <p>render this</p>
{{ end }}
```

Blocks are closed with a `{{ end }}`

When the `if` binding is rendered, it will run the ruby code after "if". If the code is true, it will render the code in the block below. Any changes to the data in the `if` condition will update the rendered block.

If bindings can also have `elsif` and `else` blocks.

```
{{ if _condition_1? }}
  <p>condition 1 true</p>
{{ elsif _condition_2? }}
  <p>condition 2 true</p>
{{ else }}
  <p>neither true</p>
{{ end }}
```

Each binding

For iteration over objects, you can use `.each`

```
{{ _items.each do |item| }}
  <p>{{ item }}</p>
{{ end }}
```

Above, if `_items` is an array, the block will be rendered for each item in the array, setting `item` to the value of the array element.

You can also call .each_with_index to place the index in a local variable within the block

```
{{ _items.each_with_index do |item, index| }}
  <p>{{ index }}. {{ item }}</p>
{{ end }}
```

For the array: `['one', 'two', 'three']` this would print:

```
0. one
1. two
2. three
```

You can do `{{ index + 1 }}` to correct the zero offset.

When items are removed or added to the array, the `each` binding automatically and intelligently adds or removes the items from/to the DOM.

Attribute Bindings

Bindings can also be placed inside of attributes.

```
<p class="{{ if _is_cool? }}cool{{ end }}">Text</p>
```

There are some special features provided to make elements work as "two way bindings":

```
<input type="text" value="{{ _name }}" />
```

CheckBoxes

In the example above, if `_name` changes, the field will update, and if the field is updated, `_name` will be changed:

```
<input type="checkbox" checked="{{ _checked }}" />
```

If the value of a checked attribute is `true`, the checkbox will be shown checked. If it's checked or unchecked, the value will be updated to `true` or `false` respectively.

Radio Buttons

Radio buttons bind to a checked state as well, except instead of setting the value to true or false, they set it to a supplied field value.

```
<input type="radio" checked="{{ _radio }}" value="one" />
<input type="radio" checked="{{ _radio }}" value="two" />
```

When a radio button is checked, whatever checked is bound to is set to the field's value. When the checked binding value is changed, any radio buttons where the binding's value matches the fields value are checked. NOTE: This seems to be the most useful behavior for radio buttons.

Select Boxes

Select boxes can be bound to a value (while not technically a DOM property, this is another convenient behavior Volt adds).

```
<select value="{{ _rating }}">
  <option value="1">*</option>
  <option value="2">**</option>
  <option value="3">***</option>
  <option value="4">****</option>
  <option value="5">*****</option>
</select>
```

When the selected option of the select above changes, `_rating` is changed to match. When `_rating` is changed, the selected value is changed to the first option with a matching value. If no matching values are found, the select box is unselected.

Template Bindings

All `views/*.html` files are templates that can be rendered inside of other views using the template binding.

```
{{ template "header" }}
```

The string passed to `template` should be a *template path*. Both templates and tags (which we'll cover later) lookup views and controllers in the same way.

Everyone wishes that we could predict the scope and required features for each part of an application, but in the real world, things we don't expect to grow large often do and things that we think will be large don't always end up that way. Templates and tags let you quickly setup reusable code/views. The location of templates or tags code can be moved as they grow without changing the way they are invoked.

Lets take a look at example lookup paths for a sample template.

```
{{ template "header" }}
```

Given the string "header", Volt will search for the view file in the following locations (in order):

Section	View File	View Folder	Component
header			
:body	header.html		
:body	index.html	header	
:body	index.html	index	header
:body	index.html	index	gems/header

Once a view is found, the associated controller will be loaded first.

Each part is explained below:

1. section Views are composed of sections. Sections start with a `<:SectionName>` and are not closed. Volt will look first for a section in the same view.

2. views Next, Volt will look for a view file with the template path. If found, it will render the body section of that view.

3. view folder Failing above, Volt will look for a view folder with the control name, and an index.html file within that folder. It will render the :body section of that view. If a controller exists for the view folder, it will make a new instance of that controller and render in that instance.

4. component Next, all folders under app/ are checked. The view path looked for is `{component}/index/index.html` with a section of :body.

5. gems Lastly, the app folder of all gems that start with `volt` are checked. They are checked for similar paths to component, above.

When you create a template binding, you can also specify multiple parts of the search path in the name. The parts should be separated by a `/`. For example:

```
    {{ template "blog/comments" }}
```

The above would search the following:

Section	View File	View Folder	Component
:comments	blog.html		
:body	comments.html	blog	
:body	index.html	comments	blog
:body	index.html	comments	gems/blog

Once the view file for the control or template is found, Volt will look for a matching controller. If the template file does not have an associated controller, a new `ModelController` will be used. Once a controller is found and loaded, a corresponding "action" method will be called on it if it exists. Action methods default to "index" unless the component or template path has two parts, in which case the last part is the action.

Escaping

When you need to use {{ and }} outside of bindings, anything in a triple mustache will be escaped and not processed as a binding:

```
{{{ bindings look like: {{this}}  }}}
```

Models

Volt's concept of a model is slightly different from the many frameworks where a model is the name for the ORM to the database. In Volt, a model is a class where you can store data easily. Models can be created with a "Volt::Persistor", which is responsible for storing the data in the model somewhere. Models created without a persistor simply store the data in the class's instance. Lets first see how to use a model.

Volt comes with many built-in models; one is called `page`. If you call `#page` on a controller, you will get access to the model.

```
page._name = 'Ryan'
page._name
# => 'Ryan'
```

Models act like a hash that you can access with getters and setters that start with an underscore. If an attribute is accessed that hasn't yet been assigned, you will get back a "nil model". Prefixing with an underscore makes sure that we don't accidentally try to call a method that doesn't exist. Instead, we get back a nil model instead of raising an exception. Fields behave similarly to a hash, but with different access and assignment syntax.

Models also let you nest data without having to manually create the intermediate models:

```
page._settings._color = 'blue'
page._settings._color
# => @'blue'

page._settings
# => @#<Volt::Model:_settings {:color=>"blue"}>
```

Nested data is automatically setup when assigned. In this case, `page._settings` is a model that is part of the page model. This allows nested models to be bound in a binding without the need to setup the model before use.

In Volt models, plural properties return a `Volt::ArrayModel` instance. ArrayModels behave the same way as normal arrays. You can add/remove items to the array with normal array methods like `#<<`, `push`, `append`, `delete`, `delete_at`, etc.

```
page._items
# #<Volt::ArrayModel:70303686333720 []>

page._items << {name: 'Item 1'}

page._items
# #<Volt::ArrayModel:70303686333720 [<Volt::Model:70303682055800 {:name=>"Item 1"}>]>

page._items.size
# => 1

page._items[0]
# => <Volt::Model:70303682055800 {:name=>"Item 1"}>
```

Nil Models

As a convenience, calling something like `page._info` returns what's called a `NilModel` (assuming it isn't already initialized). NilModels are placeholders for possible future Models. NilModels allow us to bind deeply-nested values without having to initialize intermediate values.

```
page._info
# => <Volt::Model:70260787225140 nil>

page._info._name
# => <Volt::Model:70260795424200 nil>

page._info._name = 'Ryan'
# => <Volt::Model:70161625994820 {:info => <Volt::Model:70161633901800 {:name => "Ryan"}>}>
```

One gotcha with NilModels is that they are a truthy value (since only nil and false are falsy in Ruby). To make things easier, calling `.nil?` on a NilModel will return true.

One common place we use a truthy check is in setting up default values with `||` (logical or). Volt provides a convenient method that does the same thing `#or`, but that works with NilModels.

Instead of:

```
    a || b
```

simply use:

```
    a.or(b)
```

Additionally, `#and` works the same way as `&&`. `#and` and `#or` let you easily deal with default values involving NilModels.

-- TODO: Document .true? / .false?

Provided Collections

Above, we mentioned that Volt comes with many default collection models accessible from a controller. Each stores in a different location.

Name	Storage Location
page	page provides a temporary store that only lasts for the life of the page.
store	store syncs the data to the backend database and provides query methods.
local_store	values will be stored in the local_store
params	values will be stored in the params and URL. Routes can be setup to change how params are shown in the URL. (See routes for more info)
flash	any strings assigned will be shown at the top of the page and cleared as the user navigates between pages.
cookies	saves as a cookie, only assign properties directly, not sub collections.
controller	a model for the current controller

more storage locations are planned

Store Collection

The store collection backs data in the data store. Currently, the only supported data store is Mongo. (More coming soon, RethinkDb will probably be next). You can use `store` very similarly to the other Volt collections.

In Volt, you can access `store` on the front-end and the back-end. Data will automatically be synced between the front-end and the backend. Any changes to the data in `store` will be reflected on any clients using the data (unless a buffer is in use - see below).

```
store._items << {name: 'Item 1'}

store._items[0]
# => <Volt::Model:70303681865560 {:name=>"Item 1", :_id=>"e6029396916ed3a4fde84605"}>
```

Inserting into `store._items` will create an `_items` table and insert the model into it. A pseudo-unique `_id` will be generated automatically.

Currently, one difference between `store` and other collections is that `store` does not store properties directly. Only ArrayModels are allowed directly on the `store` collection.

```
store._something = 'yes'
# => won't be saved at the moment
```

Note: We're are planning to add support for direct `store` properties.

Store Model State

Because there is a delay when syncing data to the server, store models provide a `state` method that can be used to determine if the model is loaded or synced.

state	description
not_loaded	data is not loaded
loading	model is fetching data from the server
loaded	data is loaded and no changes are unsynced
dirty	data has been changed, but is not synced back to the server yet
inactive	model is not listening for updates.

The `inactive` state can happen because there are no current event listeners or bindings listening on the model.

Flash Collection

The flash collection lets you easily display information to the user on the client side. Flash contains four (default) subcollections, `successes`, `notices`, `warnings`, and `errors`. When these collections are appended to, the message will be displayed in a div with the classes of `"alert alert-{{ ..collection name.. }}"`

Example

```
flash._successes << "Your data has been saved"
```

```
flash._errors << "Unable to save because you're not on the internet"
```

Strings added to any subcollection on flash will be removed after 5 seconds. By default the flash message can be clicked to clear.

Local Store Collection

The `local_store` collection persists its data in the browser's local store.

Cookies Collection

The `cookies` collection stores data in a browser cookie. Each assigned property gets saved to a cookie of the same name:

```
cookies._user_id = 520

puts cookie._user_id
# => "520"

cookies.delete(:user_id)
```

Values in the cookie collection are converted to strings. Adding expiration and other options are still on our todo list. Right now cookies default to a 1 year expiration.

Sub Collections

Models can be nested on `store`:

```
store._states << {name: 'Montana'}
montana = store._states[0]

montana._cities << {name: 'Bozeman'}
montana._cities << {name: 'Helena'}

store._states << {name: 'Idaho'}
idaho = store._states[1]

idaho._cities << {name: 'Boise'}
idaho._cities << {name: 'Twin Falls'}

store._states
# #<Volt::ArrayModel:70129010999880 [<Volt::Model:70129010999460 {:name=>"Montana", :_id=>"e3aa44651ff2e705b8f8319e
```

You can also create a Model first and then insert it:

```
montana = Model.new({name: 'Montana'})

montana._cities << {name: 'Bozeman'}
montana._cities << {name: 'Helena'}

store._states << montana
```

Model Classes

By default all collections use the `Volt::Model` class.

```
page._info.class
# => Volt::Model
```

You can provide classes that will be loaded in place of the standard model class. Model classes should inherit from `Volt::Model`. You can place these in any app/{component}/models folder. For example, you could add `app/main/info.rb`:

```
class Info < Volt::Model
end
```

Now when you access any sub-collection called `_info`, it will load as an instance of `Info`

```
page._info.class
# => Info
```

This lets you set custom methods and validations within collections.

Fields

Once you have a model class you can specify fields on the model explicitly instead of using the underscore syntax.

```ruby
class Post < Volt::Model
  field :title
  field :body, String
end
```

Fields can optionally take a type restriction. Once you add fields, they can be read and assigned with getter and setter methods.

```ruby
new_post = Post.new(body: 'it was the best of times')

new_post.title = 'A Title'

new_post.title
# => 'A Title'

store._posts << new_post
```

Buffers

Because the store collection is automatically synced to the backend, any change to a model's property will result in all other clients seeing the change immediately. Often this is not the desired behavior. To facilitate building CRUD apps, Volt provides the concept of a "buffer". A buffer can be created from one model and will not save data back to its backing model until .save! is called on it. This lets you create a form thats not saved until a submit button is pressed.

```
store._items << {name: 'Item 1'}

item1 = store._items[0]

item1_buffer = item1.buffer

item1_buffer._name = 'Updated Item 1'
item1_buffer._name
# => 'Updated Item 1'

item1._name
# => 'Item 1'

item1_buffer.save!

item1_buffer._name
# => 'Updated Item 1'

item1._name
# => 'Updated Item 1'
```

`#save!` on buffer also returns a promise that will resolve when the data has been saved back to the server.

```
item1_buffer.save!.then do
  puts "Item 1 saved"
end.fail do |err|
  puts "Unable to save because #{err}"
end
```

Calling .buffer on an existing model will return a buffer for that model instance. If you call .buffer on a Volt::ArrayModel (plural sub-collection), you will get a buffer for a new item in that collection. Calling .save! will then add the item to that sub-collection as if you had done << to push the item into the collection.

Validations

Within a model class, you may setup validations. Validations let you restrict the types of data that can be stored in a model. Validations are mostly useful for the `store` collection, though they can be used elsewhere.

At the moment, we only have two validations implemented (length and presence). A lot more are coming!

```
class Info < Volt::Model
  validate :_name, length: 5
  validate :_state, presence: true
end
```

When calling save on a model with validations, the following occurs:

1. Client side validations are run; if they fail, the promise from `save!` is rejected with the error object.
2. The data is sent to the server where client- and server-side validations are both run on the server; any failures are returned and the promise is rejected on the front-end (with the error object)
    - re-running the validations on the server-side helps to make sure that no data can be saved that doesn't pass the validations.
3. If all validations pass, the data is saved to the database and the promise is resolved on the client.
4. The data is synced to all other clients.

ArrayModel Events

Models trigger events when their data is updated. Currently, models emit two events: `added` and `removed`. For example:

```
model = Volt::Model.new

model._items.on('added') { puts 'item added' }
model._items << 1
# => item added

model._items.on('removed') { puts 'item removed' }
model._items.delete_at(0)
# => item removed
```

Automatic Model Conversion

Hash -> Model

For convenience, when placing a hash inside of another model, it is automatically converted into a model. Models are similar to hashes, but provide support for things like persistence and triggering reactive events.

```
user = Volt::Model.new
user._name = 'Ryan'
user._profiles = {
  twitter: 'http://www.twitter.com/ryanstout',
  dribbble: 'http://dribbble.com/ryanstout'
}

user._name
# => "Ryan"
user._profiles._twitter
# => "http://www.twitter.com/ryanstout"
user._profiles.class
# => Volt::Model
```

Models are accessed differently from hashes. Instead of using `model[:symbol]` to access, you call a method `model.method_name`. This provides a dynamic unified store where setters and getters can be added without changing any access code.

You can get a Ruby hash back out by calling `#to_h` on a Model.

Array -> ArrayModel

Arrays inside of models are automatically converted to an instance of ArrayModel. ArrayModels behave the same as a normal Array except that they can handle things like being bound to backend data and triggering reactive events.

```
model = Volt::Model.new
model._items << {name: 'item 1'}
model._items.class
# => Volt::ArrayModel

model._items[0].class
# => Volt::Model
model._items[0]
```

To convert a Volt::Model or a Volt::ArrayModel back to a normal hash, call .to_h or .to_a respectively. To convert them to a JavaScript Object (for passing to some JavaScript code), call `#to_n` (to native).

```
user = Volt::Model.new
user._name = 'Ryan'
user._profiles = {
  _twitter: 'http://www.twitter.com/ryanstout',
  _dribbble: 'http://dribbble.com/ryanstout'
}

user._profiles.to_h
# => {twitter: 'http://www.twitter.com/ryanstout', dribbble: 'http://dribbble.com/ryanstout'}

items = Volt::ArrayModel.new([1,2,3,4])
# => #<Volt::ArrayModel:70226521081980 [1, 2, 3, 4]>

items.to_a
# => [1,2,3,4]
```

You can get a normal array again by calling .to_a on a Volt::ArrayModel.

Controllers

A controller can be any class in Volt. However, it is common to have that class inherit from `Volt::ModelController`. A model controller lets you specify a model that the controller works off of. This is a common pattern in Volt. The model for a controller can be assigned by one of the following:

1. A symbol representing the name of a provided collection model:

```
class TodosController < Volt::ModelController
  model :page

  # ...
end
```

1. Calling `self.model=` in a method:

```
class TodosController < Volt::ModelController
  def initialize
    self.model = :page
  end
end
```

When a model is set, any missing methods will be proxied to the model. This lets you bind within the views without prefixing the model object every time. It also lets you change out the current model and have the views update automatically.

In ModelControllers, the `#model` method returns the current model.

See the provided collections section for a list of the available collection models.

You can also provide your own object to be a model.

In the example above, any methods not defined on the TodosController will fall through to the provided model. All views in `views/{controller_name}` will have this controller as the target for any Ruby run in their bindings. This means that calls on `self` (implicit or with `self`.) will have the model as their target (after calling through the controller). This lets you add methods to the controller to control how the model is handled, or provide extra methods to the views.

Volt is more similar to an MVVM architecture than an MVC architecture. Instead of the controllers passing data off to the views, the controllers are the context for the views. When using a `Volt::ModelController`, the controller automatically forwards all methods it does not handle to the model. This is convenient since you can set a model in the controller and then access its properties directly with methods in bindings. This lets you do something like `{{ _name }}` instead of something like `{{ model._name }}`

Controllers in the app/main component do not need to be namespaced, all other components should namespace controllers like so:

```
module Auth
  class LoginController < Volt::ModelController
    # ...
  end
end
```

Here "auth" would be the component name.

Reactive Accessors

The default `Volt::ModelController` proxies any missing methods to its model. Sometimes you need to store additional data reactively in the controller outside of the model. (Though often you may want to consider doing another control/controller). In this case, you can add a `reactive_accessor`. These behave just like `attr_accessor` except that the values assigned and returned are tracked for any Computations.

```ruby
class Contacts < Volt::ModelController
  reactive_accessor :query
end
```

Now, from the view, we can bind to `query` while also changing in and out the model. You can also use `reactive_reader` and `reactive_writer`. When `query` is accessed it tracks that it was accessed and will rerun any Computations when it changes.

url_for and url_with

If you need to generate urls using the routes, you can call `url_for` or `url_with`

url_for

`url_for` takes a hash of params and returns a url based on the routes and passed in params.

Below is an example of doing a link to change `?page=` on the query string. This example assumes routes exist for a todos controller.

```
url_for(controller: 'todos', page: 5)
# => 'http://localhost:3000/todos?page=5'
```

url_with

`url_with` is like `url_for`, but merges in the current params. In the example below, assume params is `{controller: 'todos'}`

```
url_with(page: 5)
# => 'http://localhost:3000/todos?page=5'
```

Because `url_for` is a controller method, it can also be accessed in views:

```
<a href="{{ url_with(page: 5) }}">page 5</a>
```

Controller Actions

When a template binding or tag renders a view, it first loads a controller. There are four callbacks around the rendering. The `{action}` is the same as the view file's name. So if you were rendering `about.html`, `about` would be the action. Simply create the correct method in the controller and it will called at the time.

action name	description
`{action}`	called before anything renders. Setup data you need
`{action}_ready`	called after the view renders. Run any code where you need to bind directly to the dom. jQuery setup code for example (bootstrap components)
`before_{action}_remove`	called before the view is removed (unrendered). Cleanup any dom bindings here.
`after_{action}_remove`	called after the view is removed from the dom. Cleanup anything in the controller that needs to be cleaned up.

Most of the time all you will need is the action method to setup the controller, model, etc...

Tasks

Sometimes you need to explicitly execute some code on the server. Volt solves this problem through *tasks*. You can define your own tasks by inheriting from `Volt::TaskHandler`. Ruby files in a `tasks` folder, which end with `_tasks.rb`, will be required automatically.

```
# app/main/tasks/logging_tasks.rb

class LoggingTasks < Volt::TaskHandler
  def log(message)
    puts message
  end
end
```

Volt will automatically generate wrappers for you on the client side which will return a promise.

Note that the classes on the server side use instance methods while the wrapper classes represent those methods as class methods For more information on using promises in ruby see here.

```
class Contacts < Volt::ModelController
  def hello
    promise = LoggingTasks.log('Hello World!')
  end
end
```

You can use the `#then` method of the returned promise to get the result of the call. You can use the `#fail` method on the promise to get any thrown errors.

```
MathTasks.add(23, 5).then do |result|
  # result should be 28
  alert result
end.fail do |error|
  puts "Error: #{error}"
end
```

Components

Apps are made up of Components. Each folder under `app/` is a component. When you visit a route, it loads all of the files in the component on the front end, so new pages within the component can be rendered without a new http request. If a URL is visited that routes to a different component, the request will be loaded as a normal page load and all of that component's files will be loaded. You can think of components as the "reload boundary" between sections of your app.

Dependencies

You can also use controls (see below) from one component in another. To do this, you must require the source component from the component you wish to use them in. This can be done in the `config/dependencies.rb` file. Just put

```
component 'component_name'
```

in the file.

Dependencies act just like `require` in ruby, but for whole components.

Sometimes you may need to include an externally hosted JS file from a component. To do this, simply do the following in the dependencies.rb file:

```
javascript_file 'http://code.jquery.com/jquery-2.0.3.min.js'
css_file '//netdna.bootstrapcdn.com/bootstrap/3.0.3/css/bootstrap.min.css'
```

Note, however, that jquery and bootstrap are currently both included by default. Dependencies included with `javascript_file` and `css_file` will be mixed in with your component assets at the correct locations according to the order that they occur in the dependencies.rb files.

Assets

Note, asset management is still early, and likely will change quite a bit

In Volt, assets such as JavaScript and CSS (or sass) are automatically included on the page for you. Anything placed inside of a components asset/js or assets/css folder is served at /assets/{js,css} (via Sprockets). Link and script tags are automatically added for each css and js file in assets/css and assets/js respectively. Files are included in their lexical order, so you can add numbers in front if you need to change the load order.

Any JS/CSS from an included component or component gem will be included as well. By default bootstrap is provided by the volt-bootstrap gem.

Note: asset bundling is on the TODO list

Component Generator

Components can easily be shared as a gem. Volt provides a scaffold command for creating component gems. In a folder (not in a volt project), simply type: `volt gem {component_name}`. This will create the files needed for a new gem. Note that all volt component gems will be prefixed with `volt-` so that they can easily be found by others on github and rubygems.

While developing, you can use the component by placing the following in your Gemfile:

```
gem 'volt-{component_name}', path: '/path/to/folder/with/component'
```

Once the gem is ready, you can release it to rubygems with:

```
rake release
```

Remove the `path:` option in the gemfile if you wish to use the rubygems version.

Provided Components

Volt provides a few components to make web developers' lives easier.

Notices

Volt automatically places `<:volt:notices />` into views. This component shows notices for the following:

1. flash messages
2. connection status (when a disconnect happens, lets the user know why and when a reconnect will be attempted)
3. page reloading notices (in development)

Flash

As part of the notices component explained above, you can append messages to any collection on the flash model.

Each collection represents a different type of "flash". Common examples are `_notices`, `_warnings`, and `_errors`. Using different collections allows you to change how the flash is displayed. For example, you might want `_notices` and `_errors` to show with different colors.

```
flash._notices << "message to flash"
```

These messages will show for 5 seconds, then disappear (both from the screen and the collection).

Tags

Tags (formerly called controls) let you render a view/controller similar to how you would with a template binding. The main difference is you can pass in attributes as arguments.

Tags start with a `:` (colon) to differentiate them from normal html tags. Like normal html tags, they should be closed.

```
<:tag-name />
```

or

```
<:tag-name></:tag-name>
```

Refer to the Template Binding section to see how tags lookup their associated view files. The above has the same lookup as `{{ template "tag-name" }}`. Doing `<:blog:comments />` is the same as `{{ template "blog/comments" }}`

A tag loads the same as a template, loading the controller, calling the action method (if one exists), and rendering the view. You can also pass in attributes to Tags.

Tag Arguments/Attributes

Like other html tags, controls can be passed attributes. These attributes are then converted into an object that is passed as the first argument to the initialize method of the controller. The standard Volt::ModelController's initialize will then assign the object to the attrs property which can be accessed with `#attrs`. This makes it easy to access attributes passed in.

```
<:Body>

  <ul>
    {{ _todos.each do |todo| }}
      <:todo name="{{ todo._name }}" />
    {{ end }}
  </ul>

<:Todo>
  <li>{{ attrs.name }}</li>
```

Instead of passing in individual attributes, you can also pass in a Volt::Model object with the `model` attribute and it will be set as the model for the controller.

```
<:Body>
  <ul>
    {{ _todos.each do |todo| }}
      <:todo model="{{ todo }}" />
    {{ end }}
  </ul>

<:Todo>
  <li>
    {{ _name }} -
    {{ if _complete }}
      Complete
    {{ end }}
  </li>
```

Routes

Routes in Volt are very different from traditional backend frameworks. Since data is synchronized using websockets, routes are mainly used to serialize the state of the application into the url in a pretty way. When a page is first loaded, the URL is parsed with the routes, and the params model's values are extracted and set from the URL. Later, if the params model is updated, the URL is updated based on the routes.

This means that routes in Volt have to be able to go both from URL to params and params to URL. It should also be noted that if a link is clicked and the controller/view to render the new URL is within the current component (or an included component), the page will not be reloaded, the URL will be updated via the HTML5 history API, and the params hash will update to reflect the new URL. You can use these changes in params to render different views based on the URL.

Routes file

Routes are specified on a per-component basis in the `config/routes.rb` file. Routes simply map from URL to params.

```
get "/todos", {_view: 'todos'}
```

Routes take two arguments: a path and a params hash. When a new URL is loaded and the path is matched on a route, the params will be set to the params provided for that route. The specified params hash acts as a constraint. For example, an empty hash will match any url. Any params that are not matched will be placed in the query parameters.

When the params are changed, the URL will be set to the path for the route whose params hash matches.

Route paths can also contain variables similar to bindings:

```
get "/todos/{{ _index }}", _view: 'todos'
```

In the case above, if any URL matches /todos/*, (where* is anything but a slash), it will be the active route. In this case, `params._view` would be set to 'todos', and `params._index` would be set to the incoming value in the path.

If `params._view` were 'todos' and `params._index` were present, the route would be matched.

Routes are matched from top to bottom in a routes file.

Users

Users are a core component of most web apps. To help standardize things, Volt builds the concept of users into the framework.

Note

Some features in the `users` component are still a work in process (as of Nov 9, 2014). The plan is to add omniauth support to be able login through third party services. Right now only an email or username/password option is provided.

Working with Users

Volt ships with the volt-user-tempates gem out of the box. First, we'll see how to use users, then we'll talk about how to create your own signup and login pages.

`volt-user-templates` provides signup and login templates that render via the default main route template. Volt provides routes at `/signup` and `/login` in `routes.rb`, or you can render the templates using a tag. See the volt-user-templates readme for more info.

You can access the current user model with `Volt.user`. This will return a user model if the user is logged in. If the user is not logged in, it will return `nil`.

Restricting Models

Volt provides helpers to ensure that models can only be modified by certain users.

... TODO DOCS ...

Logging In

You can log a user in with:

```
Volt.login(login, password)
```

In the above, `login` can be either the username or email based on the config. `Volt.login` returns a promise that resolves on a successful login, or fails with an error message on an unsuccessful login.

```
Volt.login(email, password).then do
  # redirect on successful login
  go '/dashboard'
end.fail do |error|
  # login failed with an error
  flash._errors << error
end
```

Logging Out

You can log a user out by calling:

```
Volt.logout
```

This call returns immediately and triggers a change event on `Volt.user`.

Creating Users

To see an example of creating users, see volt-user-templates.

Volt provides a `Volt::User` class that any model can inherit from. By default, Volt provides a User model in `app/main/models/user.rb`.

By default, `Volt::User` uses the `email` property as the login, however you can configure an app to use `username` instead. To do this, in `config/app.rb`, add:

```
config.public.auth.use_username = true
```

`Volt::User` provides validations on `email` or `username`. Passwords can be stored in the `password` property. Passwords will be hashed using bcrypt and stored in `hashed_password`. You should not need to deal with `hashed_password` directly.

To create a user, you use the normal store collection:

```
def index
  self.model = store._users.buffer
end
```

You can use volt-fields to show any errors when creating a user.

Channel

Controllers provide a `#channel` method, that you can use to get the status of the connection to the backend. Channel's access methods are reactive and when the status changes, the watching computations will be re-triggered. It provides the following:

method	description
connected?	true if it is connected to the backend
status	possible values: :opening, :open, :closed, :reconnecting
error	the error message for the last failed connection
retry_count	the number of reconnection attempts that have been made without a successful connection
reconnect_interval	the time until the next reconnection attempt (in seconds)

Testing

** Testing is being reworked at the moment.

Volt provides rspec and capybara out of the box. You can test directly against your models, controllers, etc... or you can do full integration tests via Capybara.

To run Capybara tests, you need to specify a driver. The following drivers are currently supported:

1. Phantom (via poltergeist)

`BROWSER=phantom bundle exec rspec`

1. Firefox

`BROWSER=firefox bundle exec rspec`

1. IE - coming soon

Chrome is not supported due to this issue with ChromeDriver. Feel free to go here and pester the chromedriver team to fix it.

Debugging

An in-browser REPL (like irb) is in the works. We also have source maps support, but it is currently disabled by default. To enable source maps, run:

```
MAPS=true volt s
```

This feature is disabled by default because (due to the volume of pages rendered) it slows down page rendering. We're working with the opal and sprockets teams to make it so everything is still served in one big source map file (which would show the files as they originated on disk).

Logging

Volt provides a helper for logging. Calling `Volt.logger` will return an instance of the Ruby logger. See here for more.

An example:

```
Volt.logger.info("Some info...")
```

You can change the logger like this:

```
Volt.logger = Logger.new
```

App Configuration

Like many frameworks, Volt changes some default settings based on an environment flag. You can set the Volt environment with the `VOLT_ENV` environment variable.

All files in the app's `config` folder are loaded when Volt boots. This is similar to the `initializers` folder in Rails.

Volt does its best to start with useful defaults. You can configure things, like your database and app name, in the `config/app.rb` file. The following are the current configuration options:

name	default	description
app_name	the current folder name	This is used internally for things like logging.
db_driver	'mongo'	Currently mongo is the only supported driver, more coming soon
db_name	"#{app*name}*#{Volt.env}	The name of the mongo database.
db_host	'localhost'	The hostname for the mongo database.
db_port	27017	The port for the mongo database.
compress_deflate	false	If true, will run deflate in the app server, its better to let something like nginx do this though

Volt Deployment

This chapter highlights various deployment options for Volt on popular cloud providers.

Heroku

Edit your `Gemfile` to specify Ruby version, e.g

```
source 'https://rubygems.org'

ruby "2.1.3" # specify a Ruby version

gem 'volt', '0.8.22'
```

Add a `Procfile` that uses Thin

```
web: bundle exec thin start -p $PORT -e $RACK_ENV
```

Set up your data store connection in `config/app.rb`. Below you see an example for MongoHQ. You'll need to adapt for your provider.

```
config.db_driver = 'mongo'
config.db_name = (config.app_name + '_' + Volt.env.to_s)

if ENV['MONGOHQ_URL'].present?
  config.db_uri = ENV['MONGOHQ_URL']
else
  config.db_host = 'localhost'
  config.db_port = 27017
end
```

Getting Help

Volt is still a work in progress, but early feedback is very much appreciated. Use the following methods to communicate with the developers, and someone will get back to you very quickly:

- **If you need help**: post on stackoverflow.com. Be sure to tag your question with `voltrb`.
- **If you found a bug**: post on github issues
- **If you have an idea or need a feature**: post on github issues
- **If you want to discuss Volt**: chat on gitter, someone from the volt team is usually online and happy to help with anything.

FAQ

- What runs on the client vs the server?
- Can I use jQuery (or other dom manipulating JS)

What runs on the client vs the server?

One confusing thing with frameworks that run on both the client and server is knowing what's running where. To simplify things, we'll break down the different parts of app's and where they run.

Controllers, Models, and Views

In volt, controllers, models, and views are accessable on both the client and server. Typically however, **controllers** and **views** usually run on the client only. In the future, they will run once on the server for an initial page request, to allow for faster loading.

Tasks

Tasks are interesting. They are primarially designed to allow the client side to call code on the server. Thanks to Volt's syncing models, you can write most of your app so it runs on the client. Running code on the client allows for fast reloading and sharing of data between actions. However, there are many times when you might want to call server code from the client:

- **Security**: Some data needs to be processed without sending it to the client.
- **Bandwidth**: You want to process a large amount of data to create a small result.
- **Opal incompatible gems**: Some gems are not supported by Opal, these can be easily accessed through Tasks.

Tasks provide an asynchronus interface to all methods in a task instance. All public task methods are accessable through a...

TODO: complete

Can I use jQuery (or other dom manipulating JS)

Yes! At the moment Volt ships with jQuery, though this will be removed in a future version. (Though you'll always be able to use Volt with jQuery) There's two parts to using jQuery in Volt:

1. Calling JS from Volts controller's (via Opal)
2. Using action callbacks to bind to the dom at the right times.

Calling JS from Volt

NOTE: The docs below are for 0.8.27.beta3 and above.

First lets talk about how to call JS code from Volt (via Opal) Opal has some docs on it here Typically you can just do inline JS by using backticks (`). If you want your controller method to run both on the server and the client, you'll need to only run it if your in opal. This can be accomplished as follows:

```
class PostController < Volt::ModelController
  def show_rendered
    if RUBY_PLATFORM == 'opal'
        # run some JS code
        `$('.post').setupCodeHighlighting()`
    end
  end
end
```

Using actions

Volt provides callbacks into the rendering life-cycle of views. See Callbacks and Actions for info. When managing your own dom bindings, you need to set them up after the view has rendered and remove them before the view removes.

```
class Post < Volt::ModelController
    def show_rendered
        # example setup JS
        `$('.post').setupCodeHighlighting();`
    end

    def before_show_remove
        # example teardown JS
        `$('.post').cleanupCodeHighlighting()`
    end
end
```

Getting the view DOM nodes

Often your views will be rendered multiple times on a page. In these case you will want to know which dom node's the view for this controller has. Currently Volt does not allow for id's with bindings in them. (This will change in the future) However Volt's `ModelController` provides two different ways to access the dom nodes.

```
class Post < Volt::ModelController
    def show
        self.container # returns the container node
        self.dom_nodes # returns a range of the nodes
    end
end
```

Views in Volt can have multiple dom nodes at the root level, allowing things like the following:

main.html

```
<ul>
    {{ posts.each do |post| }}
        <:post model="{{ post }}" />
    {{ end }}
</ul>
```

post.html

```
<li>{{ post._title }}</li>
<li>{{ post._date }}</li>
```

Because views do not have a single root node, `#dom_nodes` returns a JS range that you can query inside of. If you want the common root node (which may be above the nodes, as in the example above), you can call `#container`

Example Using Container

```
class Post < Volt::ModelController
    def show
        post = self.container
        `$(post).setupCodeHighlighting()`
    end
end
```

In opal, local variables compile down to JS local variables, so we can assign `post` and then call `#container`. Inside of our JS we have access to the `post` variable. We can pass the container node to jQuery and call our method on it.

Contributing

You want to contribute? Great! Thanks for being awesome! We have full docs on how to contribute here: https://github.com/voltrb/volt/blob/master/CONTRIBUTING.md