handlebars

Block helpers make it possible to define custom iterators and other functionality that can invoke the passed block with a new context.

Basic Blocks

For demonstration purposes, let's define a block helper that invokes the block as though no helper existed.

```
<div class="entry">
  <h1>{{title}}</h1>
  <div class="body">
    {{#noop}}{{body}}{{/noop}}
  </div>
</div>
```

The `noop` helper (short for "no operation") will receive an options hash. This options hash contains a function (`options.fn`) that behaves like a normal compiled Handlebars template. Specifically, the function will take a context and return a String.

```
Handlebars.registerHelper('noop', function(options) {
```

```
  return options.fn(this);
});
```

Handlebars always invokes helpers with the current context as `this`, so you can invoke the block with `this` to evaluate the block in the current context.

Any helpers defined in this manner will take precedence over fields defined in the context. To access a field that is masked by a helper, a path reference may be used. In the example above a field named `noop` on the context object would be referenced using:

```
{{./noop}}
```

Basic Block Variation

To better illustrate the syntax, let's define another block helper that adds some markup to the wrapped text.

```
<div class="entry">
  <h1>{{title}}</h1>
  <div class="body">
    {{#bold}}{{body}}{{/bold}}
  </div>
</div>
```

The `bold` helper will add markup to make its text bold. As before, the function will take a context as input and return a String.

```
Handlebars.registerHelper('bold', function(options) {
  return new Handlebars.SafeString(
      '<div class="mybold">'
      + options.fn(this)
      + '</div>');
});
```

The `with` helper

The `with` helper demonstrates how to pass a parameter to your helper. When a helper is called with a parameter, it is invoked with whatever context the template passed in.

```
<div class="entry">
  <h1>{{title}}</h1>
  {{#with story}}
    <div class="intro">{{{intro}}}</div>
    <div class="body">{{{body}}}</div>
  {{/with}}
</div>
```

You might find a helper like this useful if a section of your JSON object contains deeply nested properties, and you want to avoid repeating the parent name. The above template could be useful with a JSON like:

```
{
  title: "First Post",
  story: {
    intro: "Before the jump",
    body: "After the jump"
  }
}
```

Implementing a helper like this is a lot like implementing the `noop` helper. Helpers can take parameters, and parameters are evaluated just like expressions used directly inside `{{mustache}}` blocks.

```
Handlebars.registerHelper('with', function(context, options) {
  return options.fn(context);
});
```

Parameters are passed to helpers in the order that they are passed, followed by the options hash.

Simple Iterators

A common use-case for block helpers is using them to define custom iterators. In fact, all Handlebars built-in helpers are defined as regular Handlebars block helpers. Let's take a look at how the built-in `each` helper works.

```
<div class="entry">
  <h1>{{title}}</h1>
  {{#with story}}
    <div class="intro">{{{intro}}}</div>
    <div class="body">{{{body}}}</div>
  {{/with}}
</div>
<div class="comments">
  {{#each comments}}
    <div class="comment">
      <h2>{{subject}}</h2>
      {{{body}}}
    </div>
  {{/each}}
</div>
```

In this case, we want to invoke the block passed to `each` once for each element in the comments Array.

```
Handlebars.registerHelper('each', function(context, options) {
  var ret = "";

  for(var i=0, j=context.length; i<j; i++) {
    ret = ret + options.fn(context[i]);
  }

  return ret;
});
```

In this case, we iterate over the items in the passed parameter, invoking the block once with each item. As we iterate, we build up a String result, and then return it

we iterate, we build up a String result, and then return it.

This pattern can be used to implement more advanced iterators. For instance, let's create an iterator that creates a `<ul>` wrapper, and wraps each resulting element in an `<li>`.

```
{{#list nav}}
  <a href="{{url}}">{{title}}</a>
{{/list}}
```

You would evaluate this template using something like this as the context:

```
{
  nav: [
    { url: "http://www.yehudakatz.com", title: "Katz Got Your Tongue" },
    { url: "http://www.sproutcore.com/block", title: "SproutCore Blog" },
  ]
}
```

The helper is similar to the original `each` helper.

```
Handlebars.registerHelper('list', function(context, options) {
  var ret = "<ul>";

  for(var i=0, j=context.length; i<j; i++) {
    ret = ret + "<li>" + options.fn(context[i]) + "</li>";
  }

  return ret + "</ul>";
});
```

Using a library like underscore.js or SproutCore's runtime library could make this a bit prettier. For example, here's what it might look like using SproutCore's runtime library:

```
Handlebars.registerHelper('list', function(context, options) {
  return "<ul>" + context.map(function(item) {

    return "<li>" + options.fn(item) + "</li>";
```

```
  }).join("\n") + "</ul>";
});
```

Conditionals

Another common use-case for block helpers is to evaluate conditional statements. As with the iterators, Handlebars' built-in `if` and `unless` control structures are implemented as regular Handlebars helpers.

```
{{#if isActive}}
  <img src="star.gif" alt="Active">
{{/if}}
```

Control structures typically do not change the current context, instead they decide whether or not to invoke the block based upon some variable.

```
Handlebars.registerHelper('if', function(conditional, options) {
  if(conditional) {
    return options.fn(this);
  }
});
```

When writing a conditional, you will often want to make it possible for templates to provide a block of HTML that your helper should insert if the conditional evaluates to false. Handlebars handles this problem by providing generic `else` functionality to block helpers.

```
{{#if isActive}}
  <img src="star.gif" alt="Active">
{{else}}
  <img src="cry.gif" alt="Inactive">
{{/if}}
```

Handlebars provides the block for the `else` fragment as `options.inverse`. You do not need to check for the existence of the `else` fragment: Handlebars will detect it automatically and register a "noop" function.

```
Handlebars.registerHelper('if', function(conditional, options) {
  if(conditional) {
    return options.fn(this);
  } else {
    return options.inverse(this);
  }
});
```

Handlebars provides additional metadata to block helpers by attaching them as properties of the options hash. Keep reading for more examples.

Conditionals may also be chained by including the subsequent helper call within the else mustache.

```
{{#if isActive}}
  <img src="star.gif" alt="Active">
{{else if isInactive}}
  <img src="cry.gif" alt="Inactive">
{{/if}}
```

It is not necessary to use the same helper in subsequent calls, the `unless` helper could be used in the else portion as with any other helper. When the helper values are different, the closing mustache should match the opening helper name.

Hash Arguments

Like regular helpers, block helpers can accept an optional Hash as its final argument. Let's revisit the `list` helper and make it possible for us to add any number of optional attributes to the `<ul>`

element we will create.

```
{{#list nav id="nav-bar" class="top"}}
  <a href="{{url}}">{{title}}</a>
{{/list}}
```

Handlebars provides the final hash as `options.hash`. This makes it easier to accept a variable number of parameters, while also accepting an optional Hash. If the template provides no hash arguments, Handlebars will automatically pass an empty object (`{}`), so you don't need to check for the existence of hash arguments.

```
Handlebars.registerHelper('list', function(context, options) {
  var attrs = Em.keys(options.hash).map(function(key) {
    return key + '="' + options.hash[key] + '"';
  }).join(" ");

  return "<ul " + attrs + ">" + context.map(function(item) {
    return "<li>" + options.fn(item) + "</li>";
  }).join("\n") + "</ul>";
});
```

Hash arguments provide a powerful way to offer a number of optional parameters to a block helper without the complexity caused by positional arguments.

Block helpers can also inject private variables into their child templates. This can be useful to add extra information that is not in the original context data.

For example, when iterating over a list, you may provide the current index as a private variable.

```
{{#list array}}
  {{@index}}. {{title}}
{{/list}}
```

```
Handlebars.registerHelper('list', function(context, options) {
```

```
  var out = "<ul>", data;

  if (options.data) {
    data = Handlebars.createFrame(options.data);
  }

  for (var i=0; i<context.length; i++) {
    if (data) {
      data.index = i;
    }

    out += "<li>" + options.fn(context[i], { data: data }) + "</li>";
  }

  out += "</ul>";
  return out;
});
```

Private variables provided via the `data` option are available in all descendent scopes.

Private variables defined in parent scopes may be accessed via pathed queries. To access the `index` field of the parent iterator, `@../index` may be used.

Make sure you create a new data frame in each helper that assigns its own data. Otherwise, downstream helpers might unexpectedly mutate upstream variables.

Also ensure that the `data` field is defined prior to attempting to interact with an existing data object. The private variable behavior is condtionally compiled and some templates might not create this field.

Block Parameters

New in Handlebars 3.0, it's possible to receive named parameters from supporting helpers.

```
{{#each users as |user userId|}}
```

```
  Id: {{userId}} Name: {{user.name}}
{{/each}}
```

In this particular example, `user` will have the same value as the current context and `userId` will have the index value for the iteration.

This allows for nested helpers to avoid name conflicts that can occur with private variables.

```
{{#each users as |user userId|}}
  {{#each user.book as |book bookId|}}
    User Id: {{userId}} Book Id: {{bookId}}
  {{/each}}
{{/each}}
```

A number of builtin helpers support block parameters and any custom helper may provide them through the `blockParams` options field.

```
Handlebars.registerHelper('block-params', function() {
  var args = [],
      options = arguments[arguments.length - 1];
  for (var i = 0; i < arguments.length - 1; i++) {
    args.push(arguments[i]);
  }

  return options.fn(this, {data: options.data, blockParams: args});
});
```

```
{{#block-params 1 2 3 as |foo bar baz|}}
  {{foo}} {{bar}} {{baz}}
{{/block-params}}
```

Implements a helper that allows for named variable declarations within a given block. This example would output `1 2 3` on render.

Helpers can determine the number of block paramemters referenced by the template via the `options.fn.blockParams` field, which is an integer count. This value represents the number of block

parameters that could be referenced by the child template. Parameters beyond this cound will never be referenced and can safely be omitted by the helper if desired. This is optional and any additional parameters passed to the template will be silently ignored.

Raw Blocks

Raw blocks are available for templates needing to handle unprocessed mustache blocks.

```
{{{{raw-helper}}}}
  {{bar}}
{{{{/raw-helper}}}}
```

will execute the helper `raw-helper` without interpreting the content.

```
Handlebars.registerHelper('raw-helper', function(options) {
  return options.fn();
});
```

will render

```
{{bar}}
```