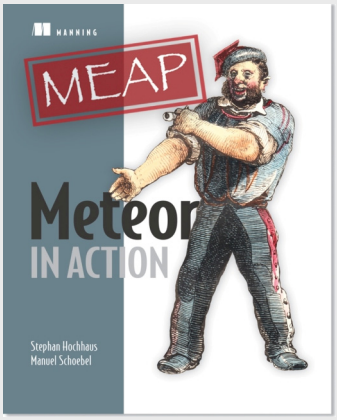

Meteor: Working with the Session object

By Stephan Hochhaus and Manuel Schoebel,

authors of *Meteor in Action*

A dedicated `Session` object that is only available on the client and lives in memory only is useful for keeping track of current user contexts and actions. In this article, we'll explore the Session object and how to use it.

Traditionally accessing a web site via HTTP is stateless. A user requests one document after another. Because there is often the need to maintain a certain state between requests, for example keep a user logged in, the most essential way to store volatile data in a web application is the session. Meteor's concept of a session is different from languages such as PHP, where a dedicated session object exists on the server or in a cookie. Meteor does not use HTTP cookies but the browser's *localStorage* instead, for example for storing session tokens to keep a user logged in.

A dedicated `Session` object that is only available on the client and lives in memory only is useful for keeping track of current user contexts and actions.

The Session object

The `Session` object holds key-value pairs, which can only be used on the client. Technically, it is a reactive dictionary that provides a `get()` and a `set()` method. Until a Session key is associated via `set()` it remains `undefined`. This can be avoided by setting a default value using `setDefault()` which works exactly as `set()`, but only if the value is currently `undefined`. As a frequent operation is to check for a session value, the `Session` object provides an `equals()` function. It is not necessary to declare a new Session variable using the `var` syntax, it becomes available as soon as a `set()` or `setDefault()` command is used.

The corresponding syntax is shown in Listing 1

Listing 1: Using the Session object

```
Session.setDefault("key", "default value"); #A
Session.get("key"); #B
Session.set("key","new value"); #C
Session.equals("key","expression"); #D
```

#A setDefault() only sets a value for a key if the key is undefined
#B returns default value
#C assigns a new value to a key
#D translates to Session.get("key") === " expression" but is more efficient

GOOD TO KNOW Although a Session variable is typically used with strings, it can also hold arrays or objects.

Let's see how we can apply the `Session` object to the housesitting app. Consider `Session` to be the app's short-term memory for keeping track of the currently selected house.

Using Session to store selected dropdown values

For the `selectHouse` template all we need to select a house from the database is a dropdown list. The idea is to retrieve all documents from the database and show all available names. Once a name is selected, it is going to define the context of all other templates and a single house is displayed. We will be using the code shown in Listing 2.

An `{{#each}}` template helper is used to iterate through all houses returned from the database. The data context is set explicitly by passing `housesNameId`[1] as an argument. Both `{{_id}}` and `{{name}}` are attributes of the `house` object coming from the database, so there is no need to define helpers for them.

Listing 2: Dropdown list code in selectHouse template

```
<template name="selectHouse">
    <select id="selectHouse">
        <option value="" {{isSelected}}></option>
        {{#each housesNameId}}#A
            <option value="{{_id}}" {{isSelected}}>{{name}}</option>
        {{/each}}
    </select>
</template>
```

#A Begin the list with an empty option to select
#B each iterates over all objects returned by a helper called housesNameId

Inside the `client.js` file, we will define a helper that provides the `housesNameId` data context. Because we haven't looked at the details of working with Collections yet we will simply return all documents and fields for now. Because `housesNameId` is defined inside a `Template` object it is reactive. This means if documents are added or removed from the

[1] For now `housesNameId` contains more than just a name and an ID, but do not worry. We will make that more efficient in a bit.

database the return value will automatically be adjusted and the select box will reflect the changes without the need to write dedicated code!

We will now use a `Session` variable called `selectedHouse` to store the dropdown selection. Since the select box should reflect the actual selection, it needs to add a `selected` attribute to the currently selected option. In order to do so, we define a second helper named `isSelected` that returns either an empty string or `selected`, if the value of `_id` equals that of our Session variable.

As the last step we need to set the value for the `Session` variable based on the user's selection. Because it involves an action coming from the user this requires an event map.

Whenever the value for the DOM element with the ID `selectHouse` changes, the event handler will set the `selectedHouse` variable to the value from the selected option element. Note that we need to pass the event as an argument to the JavaScript function that sets the `Session` value in order to access its value (see Listing 3).

Listing 3: JavaScript code for selecting houses

```
Template.selectHouse.helpers({
    housesNameId: function () {#A
        return HousesCollection.find({}, {});
    },
    isSelected: function () {#B
        return Session.equals('selectedHouse', this._id) ? 'selected' : '';
    }
});
Template.selectHouse.events = {
    'change #selectHouse': function (evt) {#C
        Session.set("selectedHouse", evt.currentTarget.value);
    }
};
```

#A returns all documents from the collection
#B returns selected if the _id for the currently processed house equals that stored inside the Session variable
#C remember to pass the event as an argument so the function can assign the selection value to the Session variable

You can test that everything works correctly by opening the JavaScript console inside a browser and selecting a value from the dropdown list. You can get and set values for the variable directly inside your console as well. If you change the value to a valid `_id` you can see that the dropdown list instantly updates itself due to the `isSelected` helper as you can see in Figure 1.

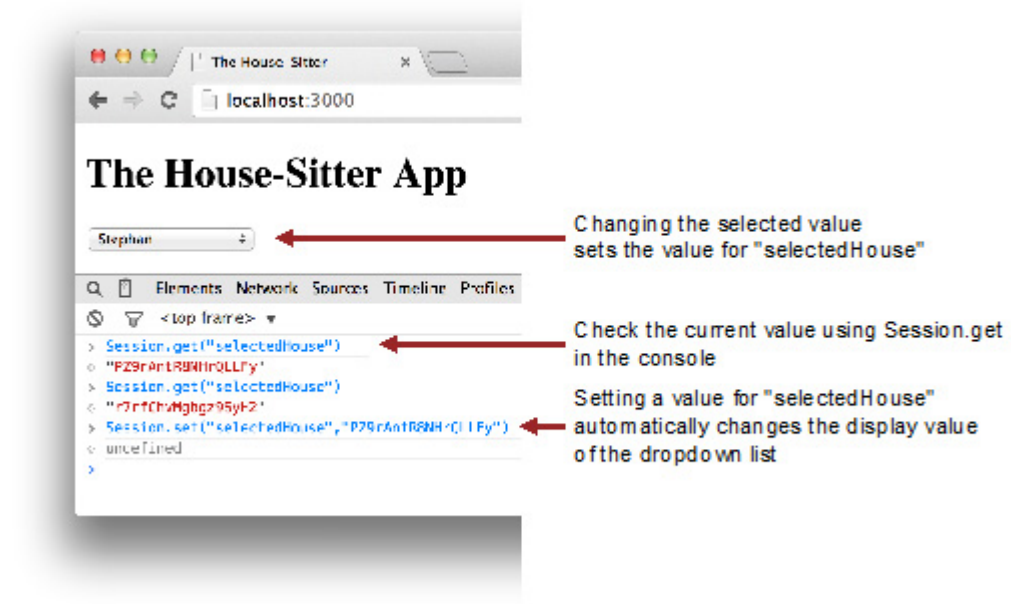

Figure 1: Getting and setting the Session variable via the JavaScript console

Creating a reactive context using Tracker.autorun

When working with JavaScript code you will often need to check for the value of a variable to better understand why an application behaves the way it does. Using `console.log()` to keep track of variable contents is one of the most important tools for debugging. Since we are dealing with reactive data sources we can also take advantage of computations to monitor the actual values of those sources. Simply put, we are going to print the contents of the reactive `Session` variable anytime it changes. In order to do so we will create a reactive context for the execution of `console.log()`.

Besides `Templates` and `Blaze` there is a third way to establish a context that enables reactive computations: `Tracker.autorun()`. Any function running inside such a block is automatically rerun whenever its dependencies (i.e., the reactive data sources used within it) change. Meteor automatically detects which data sources are used and sets up the necessary dependencies.

We can keep track of the value for `Session.get("selectedHouse")` by putting it inside an `autorun`. We place this code at the very beginning of the `client.js` file, outside of any `Template` blocks (see Listing 4). Whenever we use the drop down list to select another value,

the console immediately prints the currently selected ID. If no house is selected it will print `undefined`.

Listing 4: Using Tracker.autorun to print a Session variable to the console

```
Tracker.autorun(function() {
    console.log("The selectedHouse ID is: " +
                Session.get("selectedHouse")
               );
});
```

As you can see, the `Session` object is very simple to work with and can be extremely useful. It can be accessed from any part of the application and maintains its values even if you change source files, and Meteor reloads your application (hot code pushes). If a user initiates a page refresh all data is lost though.

Keep in mind, though, that the contents of a Session object never leave the browser, so other clients or even the server may never access its contents.