# DATA-BINDING

Двусторонняя связь данных (data-binding) — это действительно важно, так как позволяет реализовать постоянную синхронизацию JS моделей с представлением, избежать массового дублирования кода, отвечающего за его обновления, и сделать приложение удобнее. Можно выделить два метода использования этой возможности на чистом JavaScript, без фреймворков: один из них основан на революционной технологии (Object.observe), другой — на оригинальной концепции (расширения get/set).

## 1: Object.observe && DOM.onChange

Object.observe() — новичок на площадке. Это встроенная возможность JS, хотя, честно говоря, это будущая возможность, так как она предложена для ES7, но уже (!) доступна в текущей стабильной версии Chrome. Она допускает реактивные изменения объекта JS, другими словами, обратный вызов срабатывает, когда объект или его свойства изменяются.

Очевидный способ использования:

log = console.log

user = {}

Object.observe(user, function(changes){

changes.forEach(function(change) {

user.fullName = user.firstName + "" + user.lastName;

});

});

user.firstName = 'Билл';

user.lastName = 'Клинтон';

user.fullName // Билл Клинтон

Это уже само по себе довольно круто и допускает полноценное реактивное программирование в рамках JS, при этом актуальность данных поддерживается с помощью push.

// <input id="foo">

user = {};

div = $("#foo");

Object.observe(user, function(changes){

changes.forEach(function(change) {

var fullName = (user.firstName || "") + "" + (user.lastName || "");

div.text(fullName);

});

});

user.firstName = 'Билл';

user.lastName = 'Клинтон';

div.text() // Билл Клинтон

Избавившись от дублирования кода с помощью вспомогательной функции:

// <input id="foo">

function bindObjPropToDomElem(obj, property, domElem) {

Object.observe(obj, function(changes){

changes.forEach(function(change) {

$(domElem).text(obj[property]);

});

});

}

user = {};

bindObjPropToDomElem(user,'name',$("#foo"));

user.name = 'Вильям'

$("#foo").text() // Вильям

Еще один способ — привяжем DOM-элемент к значению JS. Неплохим решением будет использование плагина .change jQuery (api.jquery.com):

// <input id="foo">

$("#foo").val("");

function bindDomElemToObjProp(domElem, obj, propertyName) {

$(domElem).change(function() {

obj[propertyName] = $(domElem).val();

alert("user.name теперь "+user.name);

});

}

user = {}

bindDomElemToObjProp($("#foo"), user, 'name');

// Введите в поле ввода 'Обама'

user.name // Обама.

Для создания двусторонней связи можно комбинировать оба способа в одну функцию:

function bindObjPropToDomElem(obj, property, domElem) {

Object.observe(obj, function(changes){

changes.forEach(function(change) {

$(domElem).text(obj[property]);

});

});

}

function bindDomElemToObjProp(obj, propertyName, domElem) {

$(domElem).change(function() {

obj[propertyName] = $(domElem).val();

console.log("obj is", obj);

});

}

function bindModelView(obj, property, domElem) {

bindObjPropToDomElem(obj, property, domElem)

bindDomElemToObjProp(obj, propertyName, domElem)

}

Стоит обратить внимание на правильное взаимодействие с DOM в случае двусторонней связи данных, так как разные DOM элементы (input, div, textarea, select) по-разному отвечают на разные вызовы (text, val). Также надо помнить, что двусторонняя связь не всегда необходима: элементы, которые отвечают за отображение, редко требуют связи представление-модель, а элементы ввода редко требуют связь модель-представление.

## 2: Копнём глубже: изменение get и set

Можно сделать ещё лучше. Одним из недостатков предыдущего решения является то, что использование .change не работает для изменений, которые не вызывают событие change, например, программное изменение DOM. Вот с этим кодом обратный вызов не сработает:

$("#foo").val('Путин')

user.name // Всё ещё Обама. Упс.

Существует более радикальный подход: переопределить геттеры и сеттеры. Это выглядит не так безопасно, так как мы уже не просто наблюдаем, а собираемся менять базовую функциональность языка: чтение и запись переменных. Тем не менее, немного метапрограммирования даёт огромные возможности, что мы скоро увидим.

Что если можно было бы переопределить чтение и запись значений объектов? В конце концов, это и есть суть связи данных. Оказывается, что с помощью Object.defineProperty() можно делать именно это.

Раньше использовался нестандартизованный, устаревший способ, но сейчас есть новый, крутой и, что самое главное, стандартизованный: с помощью Object.defineProperty:

user = {}

nameValue = 'Joe';

Object.defineProperty(user, 'name', {

get: function() { return nameValue },

set: function(newValue) { nameValue = newValue; },

configurable: true // Для того, чтобы можно было переопределить это позднее

});

user.name // Джо

user.name = 'Боб'

user.name // Боб

nameValue // Боб

Хорошо, теперь user.name является алиасом свойства nameValue. Но мы можем больше, чем переадресовывать переменную — мы можем создать связь между моделью и представлением.

//<input id="foo">

Object.defineProperty(user, 'name', {

get: function() { return document.getElementById("foo").value },

set: function(newValue) { document.getElementById("foo").value = newValue; },

configurable: true // Для того, чтобы можно было переопределить это позднее

});

user.name теперь привязано к значению поля #foo. Это очень простой пример связывания (биндинга) на уровне языка с помощью определения (или расширения) нативного get/set. Этот код можно легко расширить или изменить для конкретной ситуации: связывая чтение/запись или расширяя только один из методов, например, для связывания других типов данных.

Как обычно, старайтесь избегать повторений. Приведём код к такому виду:

function bindModelInput(obj, property, domElem) {

Object.defineProperty(obj, property, {

get: function() { return domElem.value; },

set: function(newValue) { domElem.value = newValue; },

configurable: true

});

}

Использование:

user = {};

inputElem = document.getElementById("foo");

bindModelInput(user,'name',inputElem);

user.name = "Джо";

alert("Значение поля теперь "+inputElem.value) // Значение поля теперь 'Джо';

inputElem.value = "Боб";

alert("Значени user.name теперь "+user.name) // Значение модели теперь 'Боб';

Обратите внимание, что приведённый выше код до сих пор использует domElem.value и так и будет при работе с элементами <input>. Это может быть расширено внутри bindModelInput для определения типа элемента и правильного метода для изменения его значения.

## Обсуждение:

DefineProperty доступно практически в любом браузере.

Стоит упомянуть, что в приведённой выше реализации представление (view) является «последней инстанцией» (по крайней мере, в некоторой перспективе). В целом, это ничем не примечательно, так как смысл двустороннего дата-биндинга в эквивалентности данных. Тем не менее, на принципиальном уровне это может быть неудобно, а в некоторых случаях может иметь неприятные последствия, например, при удалении DOM-элемента: станет ли наша модель бесполезной? Ответ — нет, не станет. Функция bindModelInput создаёт замыкание вокруг domElem, оставляя его в памяти и сохраняя его поведение, как будто связь с моделью, даже, если DOM-элемент был удалён. Таким образом, модель продолжает существовать. Обратное также верно — если модель удалена, то с представлением ничего не случится. Понимание того, как это работает изнутри, может оказаться полезным в крайних случаях: при обновлении и модели, и представления.

Такой простой подход дает некоторые преимущества перед использованием для дата-биндинга фреймворков вроде Knockout или Angular, например:

* *понимание:* раз исходный код связи данных написан собственноручно, его гораздо проще изменять и поддерживать;
* *проиводительность:* не надо связывать «мух с котлетами», а только необходимые данные, чтобы избежать проблем со скоростью работы при большом количестве наблюдаемых объектов;
* *отсутствие зависимостей*: умение связывать данные даёт огромную свободу, если вы не ограничены уже используемым фреймворком.

Одно из слабых мест заключается в том, что, так как это не "настоящее" связывание (отсутствует проверка на "грязные" свойства объекта), в некоторых случаях изменение представления не вызовет никаких изменений в модели: например, не получится синхронизировать два DOM-элемента с помощью представления. То есть, если к модели привязаны два элемента, они обновятся только после того, как модель будет тронута (touch). Это можно сделать с помощью специальной функции:

// <input id='input1'>

// <input id='input2'>

input1 = document.getElementById('input1')

input2 = document.getElementById('input2')

user = {}

Object.defineProperty(user, 'name', {

get: function() { return input1.value; },

set: function(newValue) { input1.value = newValue; input2.value = newValue; },

configurable: true

});

input1.onchange = function() { user.name = user.name } // Поля синхронизированы

## TL;DR:

Простой способ создать двустороннюю связь данных между моделью и представлением с помощью нативного Javascript:

function bindModelInput(obj, property, domElem) {

Object.defineProperty(obj, property, {

get: function() { return domElem.value; },

set: function(newValue) { domElem.value = newValue; },

configurable: true

});

}

// <input id="foo">

user = {}

bindModelInput(user,'name',document.getElementById('foo')); // Вуаля, получаем двусторонний дата-биндинг

## Источники

http://frontender.info/native\_javascript\_data\_binding/