

Getting started

- 1. How KO works and what benefits it brings
- 2. Downloading and installing

Observables

- 1. Creating *view models* with *observables*
- 2. Working with *observable arrays*

Computed observables

- 1. Using *computed observables*
- 2. *Writable* computed observables
- 3. How dependency tracking works
- 4. *Pure* computed observables
- 5. Reference

Bindings

Controlling text and appearance

- 1. The **visible** binding
- 2. The **text** binding
- 3. The **html** binding
- 4. The **css** binding
- 5. The **style** binding
- 6. The **attr** binding

Control flow

- 1. The **foreach** binding
- 2. The **if** binding
- 3. The **ifnot** binding
- 4. The **with** binding
- 5. The **component** binding

Working with form fields

- 1. The **click** binding
- 2. The **event** binding
- 3. The **submit** binding
- 4. The **enable** binding
- 5. The **disable** binding
- 6. The **value** binding
- 7. The **textInput** binding
- 8. The **hasFocus** binding
- 9. The **checked** binding
- 10. The **options** binding
- 11. The **selectedOptions** binding
- 12. The **uniqueName** binding

Rendering templates

- 1. The **template** binding

Binding syntax

- 1. The **data-bind** syntax
- 2. The binding context

Creating custom bindings

- 1. Creating custom bindings
- 2. Controlling descendant bindings
- 3. Supporting virtual elements
- 4. Custom disposal logic

# The "with" binding

## Purpose

The **with** binding creates a new binding context, so that descendant elements are bound in the context of a specified object.

Of course, you can arbitrarily nest **with** bindings along with the other control-flow bindings such as **if** and **foreach**.

## Example 1

Here is a very basic example of switching the binding context to a child object. Notice that in the **data-bind** attributes, it is *not* necessary to prefix **latitude** or **longitude** with **coords.**, because the binding context is switched to **coords**.

```
<h1 data-bind="text: city"> </h1>
<p data-bind="with: coords">
  Latitude: <span data-bind="text: latitude"> </span>,
  Longitude: <span data-bind="text: longitude"> </span>
</p>

<script type="text/javascript">
  ko.applyBindings({
    city: "London",
    coords: {
      latitude: 51.5001524,
      longitude: -0.1262362
    }
  });
</script>
```

## Example 2

This interactive example demonstrates that:

- The **with** binding will dynamically add or remove descendant elements depending on whether the associated value is **null/undefined** or not
- If you want to access data/functions from parent binding contexts, you can use special context properties such as **\$parent** and **\$root**.

Try it out:

Twitter account:

### Source code: View

```
<form data-bind="submit: getTweets">
  Twitter account:
  <input data-bind="value: twitterName" />
  <button type="submit">Get tweets</button>
</form>

<div data-bind="with: resultData">
  <h3>Recent tweets fetched at <span data-bind="text: retrievalDate"> </span>
  <ol data-bind="foreach: topTweets">
    <li data-bind="text: text"></li>
  </ol>

  <button data-bind="click: $parent.clearResults">Clear tweets</button>
</div>
```

### Source code: View model

```
function AppViewModel() {
  var self = this;
  self.twitterName = ko.observable('@example');
```

5. Preprocessing: Extending the binding syntax

Components

- 1. Overview: What *components* and *custom elements* offer
- 2. Defining and registering components
- 3. The **component** binding
- 4. Using custom elements
- 5. Advanced: Custom component loaders

Further techniques

- 1. Loading and saving JSON data
- 2. Extending observables
- 3. Deferred updates
- 4. Rate-limiting observables
- 5. Unobtrusive event handling
- 6. Using **fn** to add custom functions
- 7. Microtasks
- 8. Asynchronous error handling

Plugins

- 1. The **mapping** plugin

More information

- 1. Browser support
- 2. Getting help
- 3. Links to tutorials & examples
- 4. Usage with AMD using RequireJs (Asynchronous Module Definition)

```
self.resultData = ko.observable(); // No initial value

self.getTweets = function() {
    var name = self.twitterName(),
        simulatedResults = [
            { text: name + ' What a nice day.' },
            { text: name + ' Building some cool apps.' },
            { text: name + ' Just saw a famous celebrity eating lard. Yum.' }
        ];

    self.resultData({ retrievalDate: new Date(), topTweets: simulatedResults });

    self.clearResults = function() {
        self.resultData(undefined);
    }
}

ko.applyBindings(new AppViewModel());
```

Parameters

- Main parameter

The object that you want to use as the context for binding descendant elements.

If the expression you supply evaluates to **null** or **undefined**, descendant elements will *not* be bound at all, but will instead be removed from the document.

If the expression you supply involves any observable values, the expression will be re-evaluated whenever any of those observables change. Then, descendant elements will be cleared out, and **a new copy of the markup** will be added to your document and bound in the context of the new evaluation result.

- Additional parameters
  - None

Note 1: Using “with” without a container element

Just like other control flow elements such as **if** and **foreach**, you can use **with** without any container element to host it. This is useful if you need to use **with** in a place where it would not be legal to introduce a new container element just to hold the **with** binding. See the documentation for **if** or **foreach** for more details.

Example:

```
<ul>
  <li>Header element</li>
  <!-- ko with: outboundFlight -->
    ...
  <!-- /ko -->
  <!-- ko with: inboundFlight -->
    ...
  <!-- /ko -->
</ul>
```

The **<!-- ko -->** and **<!-- /ko -->** comments act as start/end markers, defining a “virtual element” that contains the markup inside. Knockout understands this virtual element syntax and binds as if you had a real container element.

Dependencies

None, other than the core Knockout library.