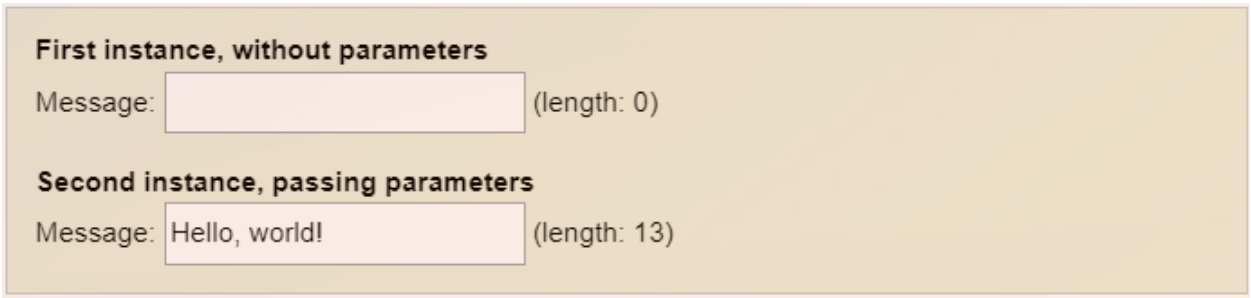
آموزش KnockoutJS ـ آموزش component binding در KnockoutJ

component مشخصی را داخل یک المان ثارد می کند و به صورت اختیاری پارامترهایی را به آن تحویل می دهد.

- Live example
- API
- Component lifecycle
- Note: Template-only components
- Note: Using component without a container element
- Note: Passing markup to components
- Disposal and memory management

مثال زنده:



Source code: View

```
<h4>First instance, without parameters</h4>
<div data-bind='component: "message-editor"'></div>

<h4>Second instance, passing parameters</h4>
<div data-bind='component: {
  name: "message-editor",
  params: { initialText: "Hello, world!" }
}'></div>
```

Source code: View model

```
ko.components.register('message-editor', {
  viewModel: function(params) {
    this.text = ko.observable(params && params.initialText || '');
```

```
  },
  template: 'Message: <input data-bind="value: text" /> '
      + '(length: <span data-bind="text: text().length"></span>)'
});

ko.applyBindings();
```

نکته: در موارد واقعی تر معمولا قالب ها و view model های کامپوننت را به جای اینکه به صورت کد داخل registration وارد کنید آن ها را از فایل های خارجی load می کنید. به این مثال و سند registration مراجعه کنید.

آموزش API در KnockoutJS

برای استفاده از component binding دو راه وجود دارد:

- **سینتکس مختصر**

در صورتی که تنها یک رشته را وارد کنید، این رشته به عنوان یک اسم کامپوننت تفسیر می شود. سپس این کامپوننت دارای اسم بدون استفاده از هیچ پارامتری در کنار آن تزریق می شود. مثال:

```
<div data-bind='component: "my-component"'></div>
```

مقدار این سینتکس می تواند قابل مشاهده هم باشد. در این صورت اگر این مقدار تغییر کند، component binding نمونه ی قدیمی کامپوننت را از بین برده و کامپوننتی که به تازگی به آن اشاره شده است را تزریق می کند. مثال:

```
<div data-bind='component: observableWhoseValueIsAComponentName'></div>
```

- **سینتکس کامل**

برای وارد کردن پارامتر ها در کامپوننت، شیئی را با مشخصات زیر ایجاد کنید:

- **name : اسم کامپوننتی است که قرار است تزریق شود. برای بار دوم، این کامپوننت می تواند قابل مشاهده باشد.**
- **params : شیئی است که به کامپوننت تحویل داده می شود. این شیء عموما یک شیء key-value بوده و پارامترهای متعددی را در خود جا میدهد و معمولا توسط component's viewmodel constructor دریافت می شود.**

مثال:

```
<div data-bind='component: {
  name: "shopping-cart",
  params: { mode: "detailed-list", items: productsList }
```

}'>

توجه داشته باشید که هر زمان کامپوننتی حذف شود (چه به این دلیل که name observable تغییر کرده باشد و چه به این خاطر که یک binding جریان کنترل کل المان را پاک کرده باشد)، کامپوننت پاک شده dispose می شود.

آموزش چرخه ی عمر کامپوننت(Life Cycle Component) در KnockoutJS

زمانی که یک component binding کامپوننتی را تزریق می کند:

1. از component loader های شما درخواست می شود تا قالب و view model factory را تامین کنند.
   - تا زمانی که اولین component loader اسم کامپوننت را تشخیص دهد، و یک view model یا قالب را تامین کند، از component loader متعددی کمک گرفته می شود. این فرآیند تنها یک بار به ازای هر نوع کامپوننت اتفاق می افتد. زیرا KnockoutJS در حافظه ی خود از تعاریف حاصل شده cache می گیرد.component loader پیش فرض بر اساس چیزی که شما ثبت کرده اید view model ها و قالب ها را تامین می کنند. در صورت امکان این مرحله یکی از ماژول های مشخص AMD را از AMD loader شما درخواست می کند.
   - به طور معمول این فرآیند یک فرآیند ناهماهنگ است. ممکن است این فرآیند شامل درخواست هایی به سرور باشد. KnockoutJS برای پایدار کردن API به صورت پیش فرض تضمین می کند که فرآیند بارگیری به صورت یک تابع callback غیر همزمان تکمیل می شود. حتی اگر این کامپوننت از قبل load شده باشد و در حافظه cache شده باشد. برای دریافت اطلاعات بیشتر در رابطه با این مطلب و چگونگی ایجاد امکان بارگیری همزمان به بخش کنترل کردن بارگیری همزمان یا غیرهمزمان مراجعه کنید.
2. قالب کامپوننت کپی شده و در المان نگهدارنده تزریق می شود.
   تمامی محتواهای موجود پاک شده و دور انداخته می شوند.
3. اگر کامپوننت دارای یک view model باشد، instantiate می شود (توسط یک نمونه معرفی می شود).
   - اگر view model به عنوان یک تابع constructor داده شود، KnockoutJS new YourViewModel(params) را فراخوانی می کند.

   اگر view model به عنوان یک تابع کارخانه (factory function) createViewModel ارائه شده باشد، KnockoutJS createViewModel(params, componentInfo) را فراخوانی می کند. که componentInfo.element المانی در درون قالبی است که هنوز مقید نشده و تزریق شده است.

   این مرحله همیشه به صورت همزمان تمام می شود (constructor ها و توابع factory نمی توانند ناهماهنگ باشند). زیرا این مرحله هر زمان که کامپوننتی instantiate می شود اتفاق می افتد و اگر بنا باشد در این مرحله منتظر درخواست های شبکه (network requests) باشیم، عملکرد چندان مطلوب نخواهد بود.
4. View model به view مقید است

- یا اینکه اگر کامپوننت view model ای نداشته باشد، در این صورت view به هر params ای که شما در اختیار component binding قرار داده اید، مقید می شود.

5. این کامپوننت فعال است
   - حالا کامپوننت در حال کار کردن است و می تواند تا هر زمان که نیاز باشد بر روی صفحه باقی بماند.

اگر یکی از پارامترهای تحویل داده شده به کامپوننت قابل مشاهده باشد، در این صورت مشخص است که کامپوننت می تواند تمامی تغییرات را مشاهده کند. یا حتی مقادیر اصلاح شده را write back کند. در این صورت است که کامپوننت می تواند به خوبی با کامپوننت مادر خود ارتباط برقرار کند. بدون اینکه نیاز باشد کد کامپوننت را به یکی از parent هایی که از آن استفاده می کند، پیوند بزنید.

6. کامپوننت نابود شده و view model ، dispose می شود
   - اگر مقدار name مربوط به component binding به طرز قابل مشاهده ای تغییر کند یا اینکه یک binding جریان کنترلی باعث حذف المان نگهدارنده شود، در این صورت تابع dispose درست پیش از اینکه المان نگهدارنده از DOM حذف شود، فراخوانی می شود.

مطلب مرتبط: مدیریت حافظه و dispose.

نکته: اگر کاربر به یک صفحه ی اینترنتی کاملا متفاوتی برود، در این صورت مرورگرها برای شلوغ نشدن کار این کار را بدون درخواست اجرای هیچ کدی انجام می دهند. به همین دلیل در این حالت هیچ تابع dispose ای احضار نمی شود. این مسئله ایرادی ندارد. زیرا مرورگر می تواند به صورت خودکار حافظه ی استفاده شده توسط تمام اشیائی که در حال کار بوده اند را رها کند.

نکته: کامپوننت های Template-only

کامپوننت ها معمولا دارای view model هستند اما الزامی برای این کار وجود ندارد. یک کامپوننت می تواند تنها یک قالب را مشخص کند.

در این حالت شیء موجود در view کامپوننت، شیء params ای است که شما به component binding انتقال داده اید. مثال:

```
ko.components.register('special-offer', {
  template: '<div class="offer-box" data-bind="text: productName"></div>'
});
```

مثال بالا را می توان با کد زیر تزریق کرد:

```
<div data-bind='component: {
   name: "special-offer-callout",
   params: { productName: someProduct.name }
}'></div>
```

یا برای راحتی کار از یک المان شخصی استفاده کرد.

```
<special-offer params='productName: someProduct.name'></special-offer>
```

نکته: استفاده از کامپوننت بدون المان نگهدارنده

در برخی مواقع ممکن است که بخواهید کامپوننتی را بدون استفاده از المان نگهدارنده ی اضافی داخل view تزریق کنید. این کار را می توانید با استفاده از سینتکس جریان کنترل بدون نگهدارنده انجام دهید. این سینتکس مبتنی بر تگ های توضیحی است. برای مثال:

```
<!-- ko component: "message-editor" -->
<!-- /ko -->
```

یا انتقال پارامترها:

```
<!-- ko component: {
  name: "message-editor",
  params: { initialText: "Hello, world!", otherParam: 123 }
} -->
<!-- /ko -->
```

کامنت های <!-- ko --> و <!-- /ko --> به عنوان نشانگرهای آغازین و پایانی عمل میکنند. به این صورت که المانی مجازی را تعریف می کنند که در داخل خود شامل markup هستند. KnockoutJS سینتکس این المان مجازی را درک کرده و درست مانند وقتی که شما یک المان نگهدارنده ی واقعی داشته باشید، مقید می شود.

نکته: انتقال markup به کامپوننت

المانی که شما به یک component binding متصل می کنید، ممکن است شامل markup های بیشتری هم باشد. برای مثال:

```
<div data-bind="component: { name: 'my-special-list', params: { items: someArrayOfPeople
} }">
  <!-- Look, here's some arbitrary markup. By default it gets stripped out
     and is replaced by the component output. -->
  The person <em data-bind="text: name"></em>
  is <em data-bind="text: age"></em> years old.
</div>
```

اگرچه گره های DOM موجود در این المان حذف می شوند(stripped out) و به صورت پیش فرض مقید نیستند، اما به این صورت هم نیست که کاملا از دسترس خارج شوند. در عوض این گره ها در کامپوننت وارد می شوند (در این حالت، my-special-list) و بعد از این کار در آینده هر زمان که نیاز باشد می توان آن ها در خروجی آن کامپوننت لحاظ کرد.

در صورتی که بخواهید کامپوننت هایی بسازید که بیانگر المان های نگهدارنده ی رابط کاربری باشد، مانند شبکه ها، لیست ها، دیالوگ ها و یا tab set ها، باید markup دلخواه را درون یک ساختار معمول تزریق کرده و آن را مقید کنید. این کار به شما کمک شایانی می کند. به "نمونه ی کامل برای المان های شخصی" مراجعه کنید. این

کار را می توانید بدون استفاده از المان های شخصی با استفاده از سینتکس نشان داده شده در بالا نیز انجام دهید.

آموزش مدیریت حافظه و Dispose در KnockoutJS

به انتخاب خودتان کلاس viewmodel شما می تواند دارای تابع dispose باشد. هر زمان که کامپوننت نابود شود و از DOM حذف شود (مثلا، چون آیتم متناظر از foreach حذف شده است یا اینکه if binding، false شده است)، KnockoutJS این تابع را فرا میخواند.

شما باید برای منتشر کردن تمامی منابعی که ذاتا قابلیت زباله روبی (garbage-collectable) ندارند از dispose استفاده کنید. برای مثال:

- توابع setInterval callback تا زمانی که کاملا پاک نشوند، به fire بودن ادامه می دهند.
  - برای متوقف کردن این callback ها از clearInterval(handle) استفاده کنید. در غیر این صورت viewmodel شما ممکن است در حافظه نگهداری شود.
- مشخصه های ko.computed تا زمانی که کاملا dispose شوند، به دریافت نوتیفیکیشن از وابستگی های خود ادامه می دهند.
  - اگر وابستگی ای در یک شیء خارجی قرار داشته باشد، در این صورت در computed property حتما باید از ()dispose. استفاده کنید. در غیر این صورت این وابستگی (و احتمالا viewmodel تان) در حافظه نگهداری خواهد شد. به عنوان جایگزین برای اینکه دیگر نیازی به dispose کردن دستی نداشته باشید می توانید از *pure* computed استفاده کنید.
- Subscriptionsهای مربوط به observable ها تا زمانی که کاملا dispose شوند، به fire بودن خود ادامه می دهند.
  - اگر در یک observable خارجی subscribe کرده اید، حتما در subscription از ()dispose. استفاده کنید. در غیر این صورت این callback (و احتمالا viewmodel تان) در حافظه نگهداری خواهند شد.
- event handler هایی که به صورت دستی در المان های خارجی DOM ایجاد شده اند، اگر داخل یک تابع createViewModel ایجاد شده باشند (یا حتی داخل یک viewmodel کامپوننت معمولی، هرچند برای اینکه بر اساس الگوی MVVM عمل کنید بهتر است این کار را انجام ندهید)، باید پاک شوند.
  - البته نیازی نیست نگران منتشر شدن event handler هایی باشید که توسط binding های استاندارد KnockoutJS در view شما ایجاد شده اند، زیرا KnockoutJS به صورت خودکار زمانی که المان ها حذف شوند این المان ها را unregister می کند.

برای مثال :

```
var someExternalObservable = ko.observable(123);
```

```
function SomeComponentViewModel() {
  this.myComputed = ko.computed(function() {
    return someExternalObservable() + 1;
  }, this);

  this.myPureComputed = ko.pureComputed(function() {
    return someExternalObservable() + 2;
  }, this);

  this.mySubscription = someExternalObservable.subscribe(function(val) {
    console.log('The external observable changed to ' + val);
  }, this);

  this.myIntervalHandle = window.setInterval(function() {
    console.log('Another second passed, and the component is still alive.');
  }, 1000);
}

SomeComponentViewModel.prototype.dispose = function() {
  this.myComputed.dispose();
  this.mySubscription.dispose();
  window.clearInterval(this.myIntervalHandle);
  // this.myPureComputed doesn't need to be manually disposed.
}

ko.components.register('your-component-name', {
  viewModel: SomeComponentViewModel,
  template: 'some template'
});
```

الزام صریحی در dispose کردن computed ها و subscription هایی که فقط به مشخصه های شیء viewmodel یکسان وابسته هستند، وجود ندارد. زیرا این کار تنها باعث ایجاد یک circular reference می شود که زباله روب های جاوا اسکریپت می دانند چگونه منتشر شوند. با این حال برای اینکه نیازی نباشد به خاطر داشته باشید که چه چیزهایی به dispose شدن نیاز دارند، بهتر است هر زمان که ممکن بود از pureComputed استفاده کنید، و همه ی computed ها و subscription های دیگر را کاملا dispose کنید. چه این کار از نظر فنی ضروری باشد یا نباشد.