# VNode Life-Cycle

## Overview

There are several distinct stages in virtual nodes life:

* Construction – VNode objects are created and their constructors are called
* Rendering – nodes produce children
* Commit – changes in nodes are applied to DOM
* Notifications – nodes are notified about certain life-cycle events

These stages are intermingled: for example, new nodes are constructed during the parent node’s rendering.

## Construction

Upon rendering, nodes are created but no any other methods are called. That means that the only code that is executed for each newly created node upon rendering is just its constructor.

Upon initial mounting, every rendered node will eventually go through the Creation life-cycle. Upon updating, the new hierarchy of nodes will be compared to the old hierarchy and many new nodes will be identical to the old counterparts. These new nodes will be dismissed. That means that constructors must do minimum work; the only work needed is to prepare the node for comparison with the old one.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Step** | **Element** | **Instance Component** | **Class Components** | **Function Component** | **Text** |
| constructor | * Remembers element name, properties and children * Extracts key | * Remembers component instance. | * Remembers component class, properties and children * Extracts key | * Remembers component function, properties and children * Extracts key | * Remembers text. |

## Creation

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Step** | **Element** | **Instance Component** | **Class Components** | **Function Component** | **Text** |
| willMount | * Parses properties into attributes and event * Removes key and ref from properties | * Sets site * Calls componentWillMount | * Creates component instance * Sets site * Sets reference * Calls componentWillMount | * Removes key and ref from properties | No action |
| render | * Returns children | * Calls component’s render | * Calls component’s render | * Calls function | No action |
| mount | * Creates element * Sets attributes and event listeners * Adds element to parent * Sets reference | No action | No action | No action | * Creates text node * Adds text node to parent |
| didMount | No action | * Calls component’s didMount | * Calls component’s didMount | No action | No action |

## Destruction

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Step** | **Element** | **Instance Component** | **Class Components** | **Function Component** | **Text** |
| willUnmount | No action | * Calls componentWillUnmount * Clears site | * Calls componentWillUnmount * Clears reference * Releases component instance | No action | No action |
| unmount | * Clears reference * Removes element from parent | No action | No action | No action | * Removes text node from parent |

## Update from updateMe

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Step** | **Element** | **Instance Component** | **Class Components** | **Function Component** | **Text** |
| render | Returns children | * Calls component’s render | * Calls component’s render | * Calls function | No action |
| Calls UpdateFromParent on children | | | | | |
| didUpdate | No action | * Calls componentDidUpdate | * Calls componentDidUpdate | No action | No action |

## Update from Parent

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Step** | **Element** | **Instance Component** | **Class Components** | **Function Component** | **Text** |
| isUpdatePossible | * Returns true if element name is the same | * Return false | * Returns true if component class is the same | * Returns true if function is the same | * Returns true |
| prepareUpdate | * Indicates that children should be updated. * Indicates that commit should be called | * Indicates that children should NOT be updated * Indicates that commit should NOT be called | * Calls shouldComponentUpdate. The result determines whether children should be updated. * Updates reference * Indicates that commit should NOT be called | * Indicates that children should be updated. * Indicates that commit should NOT be called | * Indicates that children should NOT be updated * Compares old and new text and if different indicates that commit should be called |
| render | * Returns children | * Calls component’s render | * Calls component’s render | * Calls function | No action |
| Determines node action for children (insert, update or delete) and for each child:   * For update: calls Update from Parent render phase * For insert: calls Creation render phase on new node * For replace: calls Destruction render phase for old node and Construction render phase on new node | | | | | |
| commit | * Updates * Updates attributes and event listeners * Updates reference | No action | No action | No action | * Sets new text to text node |
| For each child:   * + For update: call commit   + For insert call Creation commit phase on new node   + For replace: calls Destruction commit phase for old node and Construction comit phase on new node | | | | | |
| didUpdate | No action | * Calls component’s didUpdate | * Calls component’s didUpdate | No action | No action |

# Node-Matching Algorithm

The node matching algorithm is implemented by the buildSubNodeDispositions method.

It compares old and new chains of sub-nodes and determines what nodes should be created, deleted or updated. The result is remembered in the given VNDisp object as an array of VNDisp objects for each new sub-node and as array of old sub-nodes that should be deleted. For new sub-nodes there are two possible actions:

* Create - mount the new node
* Update - update the old node with information from the new node.

We try to match each new sub-node with an old sub-node in order to perform the Update action, which is more efficient than Delete and Create as it may reuse some or even most of DOM elements and attributes. Matching is done either by keys or by order:

* If there are a new node and an old node with the same key, the nodes match.
* Old nodes without a key or with a key, for which there is no a matching new node, are matched in respective order with the new nodes without a key or with a key, for which there is no a matching old node.
* If there are more old nodes than new nodes, the extra old nodes are unmounted.
* If there are more new nodes than old nodes, the new nodes are mounted.
* Note that even if an old and a new node match either by keys or by their sequential order, the Update action will only be performed under the following conditions:
  + Nodes must be of the same type: that is, a component node cannot update an element node.
  + Element nodes must have the same tag name: that is, a <div> cannot update a <span>.
  + Class –based and Function components must have the same class/function name.
  + Instance-based components must be of the same class: that is, have the same constructor function.

# Ordering Nodes

After new and old nodes were matched for updating, the new nodes transfer their information (e.g. new properties) to the old nodes, so that the old nodes and the DOM elements that they carry remain, while the new nodes are deleted (we are only talking about update actions – of course the new nodes with the Insert actions remain too). The order of the old nodes, however, may be completely different from the order of the matching new nodes and we need to move the old DOM nodes around to place them in the new order.

Depending on the discrepancies in the old and the new node sequences, understanding what moves should be done can be difficult. Since changing the order of an element is achieved via the call to the DOM’s insertBefore method, ordering should start from the end: this way we can be assured that the nodes that we already moved should not be moved again and thus, when we come to the start of the sequence we can be sure that the order is correct. We don’t want to issue the insertBefore call for every node in the sequence because it can be expensive.

This section provides an optimization algorithm of moving nodes around. This is not a scientific optimization and it is based on an assumption that in most cases, the ordering changes are not random but occur in groups. For example, if we have a list of items then it is unlikely that with a single operation the entire list will be suddenly completely rearranged. It is more likely that either a single item or a group of items are moved from one place in the list to another. For example:

* An item can be brought to the top or the bottom of the list.
* An item can be dragged to a certain place in the middle of the list.
* Some new items can be added to the top or bottom of the list (this is the case with virtual table while scrolling).

The first thing we do is to separate all node changes into 3 categories: old nodes to be deleted, old nodes to be updated from the new nodes and new nodes to be inserted. We first delete, then update and then insert nodes. The rest of the discussion deals mostly with updating nodes – because this is where node movement occurs. For the new nodes to be inserted, we just need to understand where in the new sequence they should be inserted.

The algorithm divides the list of new nodes into groups. Group starts with either the first node from the list or a node after the node that is out-of-order. Group ends either with the last node in the list or the first node that is out-of-order. Out-of-order nodes are nodes whose next sibling is different in the new list as opposed to the old list. Consider the following example:

* Old list: 0, 1, 2, 3, 4, 5, 6
* New list: 3, 0, 1, 2, 4, 5, 6, (moved node 3 to the top).

In the new list, the out-of-order nodes are 3 (was before 4, now before 0) and 2 (was before 3, now before 4), and the list can be divided into the following groups: [3], [0, 1, 2] and [4, 5, 6]. Let’s color the groups in both old and new sequences to make it more visual:

* Old list: 0, 1, 2, 3, 4, 5, 6
* New list: 3, 0, 1, 2, 4, 5, 6

We go over groups from the end of the new sequence. For each group, we compare the current next sibling of the group’s last element (in actual DOM) with the desired next sibling in the new list. We are comparing the actual next sibling DOM element with the nextSibling call, which is more than an order of magnitude faster than the insertBefore call.

If we understand that the group should be moved, we move all its items one-by-one. If we understand that two consecutive groups should swap places, we move the shorter of the two groups.

Let’s do the ordering for our example:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Current group** | **What to do** | **New sequence** | **# of moves** | **Comments** |
| [4, 5, 6] | Nothing | 0, 1, 2, 3, 4, 5, 6 | 0 |  |
| [0, 1, 2] | Move before [3] | 3, 0, 1, 2, 4, 5, 6 | 1 | We understand that the groups [0, 1, 2] and [3] should be swapped and move the shorter of them. |
| [3] | Nothing | 3, 0, 1, 2, 4, 5, 6 | 0 |  |

In order to understand what to do with each group we use the following algorithm:

* We iterate from the last group in the list to the second group in the list. We don’t need to iterate over the first group in the list because when we are done with the second group, all the groups have already been ordered.
* On each iteration, we work with three groups: the current group (i), the next group (i+1) and the previous group (i-1). The next group is either the group that was current in the previous iteration or the group that was moved into this place during the previous iteration. For the first iteration the next group doesn’t exist.
* We determine the actual location of the current group in the old node sequence and if it is not equal to the desired location in the new sequence, we need to move the group. By location we mean the first item of the group before which our current group is located (or, for the desired location, should be located).
* Now that we need to move the group we want to understand whether the move is actually a swapping of two consecutive groups. This is important because in this case we can move the shorter of the two groups. The swapping occurs if the group before which we need to move our current group is our previous group.

In the above discussion, when we used the word “node” we meant a Virtual Node (unless it was explicitly specified as DOM node). Note, however, that when a virtual node is for a component, the actual number of DOM nodes the component produces may be 0, 1 or more than 1. That means that when we are talking about moving nodes, we actually may need to move multiple DOM nodes.

# Code Structure

When a package is published, the files (including the .d.ts files) are taken from their places in the file system. When the package is installed, the files are put into the same places. Therefore, if we use a “dist” directory in tsconfig.json to put all the type definition files there, they will be all installed under the consumer package’s “node\_modules/<package>/dist” directory. This all would be good if not for the module augmentation mechanism. When module is augmented (by the consumer package code) the declare module sentence must reference the path to the file – not just the package. For example, the file src/core/HtmlTypes.ts contains IHtmlElementProps interface, which can be augmented in order to define custom attributes. This will be done in the following way:

import \* as mim from “mimbl”

declare module “mimbl/dist/core/HtmlTypes”  
{  
 interface IHtmlElementProps  
 {  
 …  
 }  
}

## Mimble Library

* mimbl – root project directory
* package.json – “main”: “./dist/mimbl.
* tsconfig.json – “outDir”: “./dist/”.
* webpack.config.js - path: \_\_dirname + "/dist"
* README.md
* /src – all source code
* mimbl.ts – re-exports all other API files
* /core – mim.ts, VNs, etc.
* /util – utility code that could be later put into a separate package
* /router – Router and Link components
* /dnd – Drag and Drop
* /dist
* mimbl.js - output