JSF & CLIENT SIDE

W3C

- The main international standards organization for the world wide web
- Was founded in 1994
- Any one can join (after approval)
- A member pays annual fee
  - Big player ~70000$
  - Small player ~8000$
- As of 2014 has 384 member
- Is criticized for
  - Being dominated by the big players
  - Slow pace

W3C Ratification Process

- Working Draft (WD)
- Candidate Recommendation (CR)
- Proposed Recommendation (PR)
- W3C Recommendation (REC)
- Working Group Note (NOTE)

W3C Vision – Year 2002

- XHTML is the future, not HTML
- XHTML 2.0 is not backward compatible with HTML 4.01 and XHTML 1.1
  - XForms instead of HTML Forms
  - XFrames instead of HTML frames
- Mozila and Opera presented a paper for HTML next generation but were denied
- As result WHATWG was born

WHATWG

- A community of people interested in evolving HTML and related technologies
- Founded by individuals from Apple, Firefox and Opera at 2004
- On 9 May 2007 W3C decided to adopt WHATWG's HTML5
- XHTML 2.0 is dead !!!

What is HTML5?

- A snapshot of WHATWG specification that is managed by W3C
- Offers new elements, new attributes and APIs
- De facto features are now documented
- Is designed so that old browser can safely ignore new features
- A better platform for building complex web application

HTML5 Status

- As of September 2014 HTML5 is considered Proposed Recommendation (PR)
  - The criteria – Two 100% complete and fully interoperable implementations
- WHATWG continues its work on HTML5 as a “living standard”
  - Never completed
  - Always updated and improved
  - New features are added but old functionality is not removed

What's new?

- Depends on who you ask ☺
- The W3C specification is stable
- But we as developers are interested in capabilities not specifications
- A standard is useful only if implemented widely
- Support for a feature may vary between browsers vendors
- You should always check for current feature support before using it
  - http://caniuse.com/

Interesting Features

Semantic Tags	Content Editable	New Input Types	Placeholder
FORM Validation	Local Storage	Autofocus	Videos and Audio Tags
SVG	Geolocation	Indexed DB	Web Sockets
Web Workers	History API	Drag & Drop	Offline Application
Web Font	XMLHttpRequest	DOM Selection	Canvas 2D
Media Capture	Web Messaging	File API	

History API

- Access to the browser's history is offered through the history object
  - Long before HTML5
  - Limited write operations
- Effectively as if the user pressed the back and forward buttons

```
window.history.back();      // go back
window.history.forward();   // go forward
window.history.go(3);       // go 3 pages forward

window.history.length       // number of history entries
```

- Full page reload ☹

History API – HTML5

- New APIs
  - history.pushState
  - history.replaceState
  - window.onpopstate
- The new API allows the developer to programmatically change the URL address
- No page reload ☺
- IE10+, Good browser support

history.pushState

- Suppose http://myws.com/admin/logins executes the following script

```
window.history.pushState({}, null, "state1");
```

- URL bar changes to http://mywebsite.com/admin/state1
- Browser **does not load** state1 from the server
- You may use absolute path

```
window.history.pushState({}, null, "/state1");
```

- URL changes to http://mywebsite.com/state1

pushState Parameters

- state – Any JavaScript object
  - Is associated with the new entry
  - Can be extracted later when the user navigates back/forward to this new entry
  - The browser holds a copy not a reference
- title – Not used
- url – The browser's URL
  - Must be of the same origin
  - If omitted the current URL is used

popstate Event

- Is dispatched to the window object when the active history entry changes

```
$(window).bind("popstate", function (e) {
     console.log("popstate: " + e.originalEvent.state);
});
```

- If the entry was created by pushState/replaceState the event's state property contains a copy of the original state
- Current state is also available through

```
console.log("Current state is: " + history.state);
```

State Serialization

- pushState serializes the state object
- popstate event deserializes it back
- You loose any metadata on the object
  - For example, prototype chaining
- Originally, some browser used JSON serialization
  - Cyclic references cause a runtime error
- Latest browsers use the “Structured clone algorithm”
  - Allows any structured graph to be serialized

History API Notes

- popstate event is not triggered on page load
- popstate event is not triggered when calling pushState/replaceState
- You probably want to wrap all details under navigate function
- There is no way to clear browser's history
- Use polyfills
  - HistoryJS - https://github.com/browserstate/history.js/
  - Backbone Router - http://backbonejs.org/#Router

Web Storage

- Passing data/state from one page to another is a common web task
- Common solutions
  - Query string – Limited storage
  - Session state – Does not scale well
  - Cookie – Limited storage
  - Hidden field – Reset upon GET request
  - URL – Limited storage

Web Storage API

- Designed to provide a larger, more secure, easier to use alternative to cookies
- Allows for key/value pairs to be stored and retrieved
- Is great for building offline web application
  - Web storage is accessible even when there is no internet connectivity
- Offers the following objects
  - sessionStorage
  - localStorage

Storage Interface

- Both sessionStorage and localStorage implements the same API
  - getItem(key)
  - setItem(key, value)
  - removeItem(key)
  - clear
- Key and value are strings !!!
- When passing object as a key or as a value it will first be converted to a string using toString

sessionStorage

- Not related to server side session management
- Each browser window/tab holds a sessionStorage object per origin
- Should live as long as the window/tab is alive
- When duplicating a tab the sessionStorage should be duplicated too
  - No sharing

```
sessionStorage.setItem("data", data);
```

```
var data = sessionStorage.getItem("data");
```

Serialization

- Web storage support only strings (key & value)
- Therefore, data must serialized/deserialized
- Can use JSON.stringify & JSON.parse
  - Cannot save cyclic references
  - Prototype is lost
  - String representation is expensive

```
var str = JSON.stringify(data);
localStorage.setItem("data", str);
```

```
var str = localStorage.getItem("data");
var data = JSON.parse(str);
```

localStorage

- A Storage object per origin
- Storage should not be cleared by browser
  - Only on rare conditions
- Different windows with same origin share the same loadStorage object
  - Browser should protect against concurrent access
- Exception is thrown under Incognito mode

```
localStorage.setItem("data", data);
```

```
var data = localStorage.getItem("data");
```

Disk Space

- Browser should limit the total amount
  - Specification does not mention the limit itself
  - 5MB is common
- Browser may prompt the user when quota is reached
  - Only supported on opera
- On some browsers the administrator can change the limit
  - Not from Java Script

storage Event

- Is fired when calling setItem/removeItem/clear
- Contains the following
  - key
  - oldValue
  - newValue
  - url – The url of the window that changes the data
- Event should not be fired on the window that changes the storage
  - IE disagrees

```
window.addEventListener("storage", function (e) {
     console.log(e);
}, false);
```

Web Storage Notes

- Updating a single record may be inefficient
  - Deserialize all data into memory
  - Fix memory
  - Serialize back
- API is blocking
  - UI is blocked while saving big data
- Should be careful when application is running under shared host
  - Wix.com

Indexed DB

- Web storage does not deal well with large data
  - Cannot update single record
  - Blocking API
- Indexed DB
  - Transactional database
  - Lets you store and retrieve objects
  - Objects are indexed → Better search performance
  - No schema
  - Asynchronous API

Browser Support

- Desktop
  - Firefox, Chrome, Safari for desktop
  - IE11 – Not complete
- Mobile
  - Android 4.4+
  - iPhone 8+ (8 is not released yet)
  - IE Mobile 10+ (Strange …)

Storage Limits

- Storage is limited per origin
- Firefox
  - No limit
  - Asks user after 50MB
- Chrome
  - No limit
- IE
  - 250MB limit
  - Asks user after 10MB

Database

- Has one or more object stores
- Has a name
- Has current version – Initially 0
- There may be multiple connections to a given database
- Each origin has an associated set of databases
- Is described by the IDBDatabase interface

Open Database

- Should specify a version – Default is 1
- Starts with no object store
- Can delete a whole database

```
var request = indexedDB.open("MyDB", 2);

request.onsuccess = function (e) {
     db = e.target.result;

     console.log("current version: " + db.version);
}

request.onerror = function (e) {
     console.log("open db error: " + e.target.error.message);
}

request.onupgradeneeded = function (e) {
     console.log("DB Upgrade is needed");
}
```

Object Store

- Has a unique name
- Has a list of records
- Has a list of indexes
- No schema
- Each record consists of a key and a value
- The list is sorted according to key
- Key is unique
- Is described by the IDBObjectStore interface

Create Object Store

- Creating new object store is only allowed during upgradeneeded event

```
request.onupgradeneeded = function (e) {
     console.log("DB Upgrade is needed");

     console.log("old version: " + e.oldVersion);
     console.log("new version: " + e.newVersion);

     var db = e.target.result;

     if (e.newVersion == 1) {
         console.log("Creating object store: contacts");

         var objectStore = db.createObjectStore("contacts",
                                { keyPath: "id", autoIncrement: true });

         console.log("Creating indexes");

         objectStore.createIndex("name", "name", { unique: false });
         objectStore.createIndex("email", "email", { unique: true });
     }
}
```

IDBObjectStore

- Properties
  - name
  - keypath
  - autoIncrement
  - indexNames
  - transaction
- Methods
  - get
  - add
  - put
  - delete
  - clear
  - index
  - createIndex/deleteIndex

Key

- Must be: number, string, Date or Array (not sparse)
- Sort order is according to language natural sort
- Can be
  - Extracted from the record itself: Use keypath

```
var objectStore = db.createObjectStore("contacts", { keyPath: "id" });
```

  - Generated by key generator: Use autoIncrement

```
var objectStore = db.createObjectStore("contacts", { autoIncrement: true });
```

  - Explicitly specified as part of the insertion

Read Single Record

- Use the get method
- Must specify the key to look for

```
var tran = db.transaction(["contacts"], "readonly");
tran.oncomplete = function () { }
tran.onerror = function (e) { }

var contacts = tran.objectStore("contacts");

var key = 1001;
var request = contacts.get(key);

request.onsuccess = function (e) {
     var contact = e.target.result;
     if (contact) {
         console.log("Item found: " + contact.name);
     }
     else {
         console.log("Item not found");
     }
}

request.onerror = function () {console.error("get error"); }
```

Read Range of Records

- Open a cursor
- Must call continue to get the next record

```
var contacts = getStore("contacts", "readonly");

var request = contacts.openCursor();

request.onsuccess = function (e) {
     var cursor = e.target.result;
     if (cursor) {
         var key = cursor.key;
         var value = cursor.value;
         console.log(key + ": " + value.id + ", " + value.name);

         cursor.continue();
     }
     else {
         console.log("No more entries");
     }
}

request.onerror = function () {
     console.log("error");
}
```

Read using an Index

- Retrieve the index and open a cursor

```
var range = IDBKeyRange.bound("Ori", "Roni", false, false);

var contacts = getStore("contacts", "readonly");
var index = contacts.index("name");

var request = index.openCursor(range, "prev");

request.onsuccess = function (e) {
     var cursor = e.target.result;
     if (cursor) {
         var key = cursor.key;
         var value = cursor.value;
         console.log(key + ": " + value.name + ", " + value.email);

         cursor.continue();
     }
     else {
         console.log("No more entries");
     }
}

request.onerror = function () { console.log("error"); }
```

Insert

- Must create a readwrite transaction
- Use add method
- Watch for success and completion of the transaction

```
var tran = db.transaction(["contacts"], "readwrite");

tran.oncomplete = function () {...}
tran.onerror = function (e) {...}
tran.onabort = function () {...}

var contacts = tran.objectStore("contacts");
var request = contacts.add({id: 1001, name: "Ori",email: "ori@gmail.com"});

request.onsuccess = function (e) {
     var key = e.target.resolve;};

request.onerror = function (e) {
     console.error("add error " + name + ": " + e.target.error.message);
};
```

No commit ?

- indexedDB supports auto committed transaction
- There is no method named commit ☺
- There is a method named abort
- So, when is transaction committed ?
  - When there are no pending change requests on the current transaction
  - And the thread returns to the browser's message loop
- Therefore,
  - You cannot postponed transaction commit
  - Once a transaction finishes any manipulation on it causes a runtime error

Concurrent Transaction

- Two readwrite transactions with overlapping scope block each other
  - Two read transaction do not block
  - Concurrent read and write transaction do not block
- The first created transaction must be completed and only then the second can be completed too
- This does not mean that your code blocks ☺
  - The onXXX are postponed until the transaction can be completed

Transaction events

- onerror is raised for every single operation that fails during the transaction
  - Use e.target.error for more details
- onabort is raised only once
- oncomplete is raised only once

```
var tran = db.transaction(["contacts"], “readwrite");

tran.oncomplete = function () {
     console.log("tran complete");
}

tran.onerror = function (e) {
     console.error("tran error: " + e.target.error.message);
}

tran.onabort = function () {
     console.error("tran abort");
}
```

Aborting Transaction

- Is aborted automatically if one of the change operation fails
- Can be aborted manually using IDBTransaction.abort
- Can throw an exception from one of the change onsuccess handlers
- Once oncomplete is raised the transaction is considered completed and cannot be aborted

Update

- There is no strict update method
- The put method updates or inserts an object

```
var tran = db.transaction(["contacts"], "readwrite");

tran.oncomplete = function () {}
tran.onerror = function (e) {}

var contacts = tran.objectStore("contacts");

var contact = { id: 1001, name: "XXX“ };

var request = contacts.put(contact);

request.onsuccess = function () {
     console.log("put success");
};

request.onerror = function () {
     console.log("put error");
};
```

Delete

- Must retrieve the object key
- Use the delete method

```
var tran = db.transaction(["contacts"], "readwrite");

tran.oncomplete = function () {...}
tran.onerror = function (e) {...}

var contacts = tran.objectStore("contacts");

var request = contacts.delete(1);

request.onsuccess = function () {
     console.log("delete success");
};

request.onerror = function () {
     console.log("delete error");
};
```

Summary

- indexedDB offers rich API for dealing with data
- Great for storing heterogeneous objects
- Quite complex API
  - With respect to the simplicity of localStorage
- Good desktop support
- No iPhone support (yet)
  - Can be implemented using Web SQL

File API

- A standard way to interact with local files
- Offers the following objects
  - File - A reference to a file
  - FileList - A collection of File objects
  - Blob - Maniulate File's data
  - FileReader – Read file asynchronously
  - URL Scheme – Binary data inside URL
- IE10+

Getting Started

- Web script cannot access a file on the file system
- However, it can
  - Use a form file input
    - User browses for the file
    - File is accessible through the input element
  - Drag & Drop
    - User drags and drops a file on the page
    - File is accessible through the dataTransfer property of the event object

Using input field

- Use multiple attribute to allow multi file selection

```
<input type="file" multiple />
```

- Can hide the input and then programmatically trigger the dialog open

```
input[type=file] {
     display: none;
}
```

```
$(".browse").click(function () {
     fileInput.trigger("click");
});
```

- Use files property on the input element

```
$.each(fileInput[0].files, function () {
     var file = this;
     console.log(file.name + ": " + file.size);
});
```

File Object

- The files property of the input element is of type FileList
  - Which is a collection of File objects
- Each File object offers the following API
  - type – Mime type
  - name – Short file name (without path)
  - size – File size in bytes
  - lastModifiedDate – Modification date
  - slice – Returns a blob object that represents the specified byte range

FileReader

- Provides methods to read a File or Blob object into memory
- Asynchronous API
- Fires progress events
- Supported formats: Text, DataURL, binary
- Can read the whole file content or multiple slices
- IE10+

FileReader API

- Read whole file
  - readAsArrayBuffer
  - readAsText
  - readAsDataUrl
- readyState attribute – EMPTY/LOADING/DONE
- Should wait for load event
- Result is stored under event.target.result
- Concurrent reads are not allowed
  - Exception is thrown

FileReader Events

- loadstart
- progress
- abort
- error
- load
- loadend

Read Text

- Use readAsText
- You may specify an encoding

```
var fileReader = new FileReader();
fileReader.readAsText(file, "windows-1255");

fileReader.addEventListener("progress", function (e) {
    progress.text(e.target.result.length);
});

 fileReader.addEventListener("load", function (e) {
    var str = e.target.result;
     result.text(str);
});
```

Read Data Url

- Use readAsDataUrl
- You may embed the result inside an img tag

```
var fileReader = new FileReader();

fileReader.readAsDataURL(file);

fileReader.addEventListener("load", function (e) {
     //
     //  load is fired when read operation completed successfully
     //
     var str = e.target.result;
     result.text(str);
     $("img").attr("src", str);
});
```

Read ArrayBuffer

- Need to wrap the ArrayBuffer inside a Typed array object in order to access its content

```
fileReader = new FileReader();

fileReader.readAsArrayBuffer(file);

fileReader.addEventListener("load", function (e) {
     var bufArr = e.target.result;

     var byteArr = new Uint8Array(bufArr);

     console.log("load: " + byteArr.length);

     for (var i = 0; i < 1000; i++) {
         console.log("    byteArr[" + i + "]: " + byteArr[i]);
     }
});
```

Slicing

- Reading a big file into memory is problematic
  - Firefox may even crash (file > 1GB)
- Should read slices instead of whole file

```
function readNextBuffer() {
     if (pos >= file.size) {
        return;
     }

     fileReader = new FileReader();
     fileReader.readAsArrayBuffer(file.slice(pos, pos + bufSize));

     fileReader.addEventListener("load", function (e) {
         pos += e.target.result.byteLength;

         var percentCompleted = Math.round(pos / file.size * 10000) / 100;

         progress.text(percentCompleted + "%");

         readNextBuffer();
     });
}
```

Summary

- Cannot access file system directly
- User should browse to the file or drop it
- File metadata is accessible using a File object
- Reading a file content is done using a FileReader object
- Content can be read as: Text, DataUrl, ArrayBuffer

Form Validation

- Validate user data without Java Script
- Is supported through a list of HTML attributes
- Can control element styling through CSS
- Can query validation status using JavaScript API
- IE10+
- No iPhone support ☹

Pattern Attribute

- All input elements can be associated with the pattern attribute
  - Text area is not supported
- The attribute expects a case sensitive Regular Expression as its value
- Empty value is not validated against the pattern
  - Use required attribute

```
<input type="text" pattern="(ab)*" />
```

Validation Lifecycle

- On page load
  - pseudo-class :invalid is attached to failed elements
- During typing
  - pseudo-class :invalid is attached/detached according to element's value
- During form submission
  - invalid DOM event is raised for every failed element
  - A popup message is displayed
  - submit event is not fired

More Validation Attributes

- required

```
<input type="text" required />
```

- maxLength – Usually is enforced during typing

```
<input type="text" maxlength="10" />
```

- min & max & step – Only for number field

```
<input type="number" min="1000" max="1200" step="2" />
```

- type – email & url

```
<input type="email" />
<input type="url" />
```

Notes

- The developer is responsible for setting CSS styling according to :invalid class
- Usually, only first failure is displayed to the user
- Message content changes between browsers
- Every browser styles its popup messages differently
- Messages are localized according to browser locale
  - Not page locale

Localizing Validation Messages

- Listen to invalid DOM event
- Reveal the reason of the failure
  - See next slides
- Register custom validation message using setCustomValidity

```
<input type="text" required class="name" />
```

```
$(".name").bind("invalid", function () {
     this.setCustomValidity("ערך חובה");
});
```

Controlling Messages Styling

- Can't really do that. Instead,
- Disable validation using novalidate attribute
  - No invalid event
  - No popup messages
  - :invalid pseudo-class is still attached
- Listen to submit event
- Call checkValidity on the form element
  - invalid event is fired
  - Check the return value
- Display validation messages the way you like

Controlling Messages Styling


```
<form novalidate>
     Name:
     <input type="text" required class="name" />

     <br />
     E-Mail:
     <input type="email" value="not valid email" />

     <br />
     <br />
     <input type="submit" value="Submit" />
</form>
```

```
$(".name").bind("invalid", function () {
     this.setCustomValidity("ערך חובה");
});

$("form").bind("submit", function () {
    if (!this.checkValidity()) {
         var invalidInputs = $("input:invalid");
         if (invalidInputs.length) {
             alert(invalidInputs[0].validationMessage);
         }
         return false;
     }
     return true;
});
```

Constraint Validation API

- Every input element has a property named validity
- Is an object of type ValidityState which supports
  - valid – True when element fails validation
  - valueMissing – required validation failed
  - patternMismatch – pattern validation failed
  - rangeOverflow/rangeUnderflow – min/max failed
  - stepMismatch – Out of possible step values
  - tooLong – maxLength validation failed. Probably never
  - typeMismatch – type email or url validation failed
  - customError – setCustomValidity was called

Summary

- FORM validation is easy
- No need to write JavaScript
- Just throw some attributes
- Probably, not suited to high-end web applications
  - No cross browser compatibility
  - Limited control over styling
  - Is not a hard task to accomplish with plain old JavaScript

Web Worker

- Traditionally,
  - JavaScript is single threaded
  - Running long computation means UI is blocked
- With HTML5 Web Worker support
  - The developer can create multiple web workers
  - Each represent a background thread
  - Long computation no longer blocks the UI
    - The developer is responsible for spawning the worker
  - What about race condition ?

Create a Worker

- Construct a new object using the Worker constructor
- Specify the URL of the script to be executed
  - Same origin policy applies
- Optionally, listen to the message event in case you want to receive messages from the worker

```
var worker = new Worker("/Scripts/Task.js");

worker.addEventListener("message", function (e) {
    ...
});
```

Thread Safety

- Web worker allows for parallel execution
- However, the worker is executed under a new global execution context
  - No sharing of global variables
- It can communicate using special channel
- All non thread safe components are unavailable when running under a worker
  - DOM
  - window
  - document

Passing Data

- Use postMessage method to send the actual data
  - From both ends
- Messages are serialized/deserialized
- This means you get a copy of the original object
- No sharing of the references
- Browsers use the structured clone algorithm
  - Cyclic references are allowed ☺

Passing Data

App.js

```
var worker = new Worker("/Scripts/Task.js");

worker.postMessage(10);

worker.addEventListener("message", function (e) {
     console.log("Result: " + e.data);
});
```

Task.js

```
self.addEventListener("message", function (e) {
     self.postMessage(e.data * 2);
});
```

Single Thread Model

- A single worker represents a single thread
- Like the browser "main" thread a web worker is not interruptible
- When a worker runs some JavaScript code it cannot process incoming messages
- Only when code finishes and returns to the browser's message loop the next queued message is processed

Terminate a Worker

- Use terminate method on the Worker object
  - Kills the worker immediately
  - The worker has no chance to complete current work

```
worker.terminate();
```

- Use close method from the Worker itself
  - Post a message from the main page
  - Close the worker

```
self.close();
```

  - close method returns and only later Worker will be closed

Unhandled Exception

- Unhandled exceptions may be tracked using the error event type
- The event can be monitored from both the worker and its creator
- Unhandled exception does not kill the worker

```
self.addEventListener("error", function (e) {
     console.log(e.message);
});
```

```
worker = new Worker("/Scripts/Task.js");

worker.addEventListener("error", function (e) {
     console.log(e.message);
});
```

Import Scripts

- All non DOM related API is accessible under web worker: Date, Math, JSON and others
- But what about your own custom code?
  - Use importScripts
- Is executed synchronously and returns only after all scripts were executed
- Url is relative to worker's script

```
importScripts("Logger.js");

Logger.message("Web worker is running and using Logger module");
```

Maximum # of Web Workers

- Specification does not mention a limit
  - Firefox has dom.workers.maxPerDomain setting which is 20 by default
  - Chrome crashes when trying to spawn 1000 web workers
  - IE has a limit of 25
- Usually, no error is reported when reaching the limit but rather the worker is queued until a previous worker is closed

Summary

- At last we have threads
- Long computation may be refactored into a background web worker
- UI may become more responsive
- However, NO DOM manipulation is allowed
  - Which means long DOM related computation still blocks the UI

Web Socket Protocol

- Enables two-way communication between browser and server
- Does not rely on opening multiple HTTP connections
- Replacement for older techniques like long polling and forever frame
- A simple abstract over TCP socket
- A totally new application protocol
  - No HTTP headers
- Managed by IETF

Web Socket API

- A JavaScript API
- Is used by the browser to initiate a Web Socket communication with the server
- Is all around the WebSocket class
  - send
  - close
  - readyState
  - onopen, onmessage, onclose, onerror
- Managed by W3C

Getting Started

- Create a new WebSocket object
- Specify URL and sub protocols (Optional)
  - Must use ws or wss protocols
- Monitor open and error events

```
var ws = new WebSocket("ws://localhost:5481/socket/connect");

ws.onerror = function (e) {
     console.log("ERROR");
     console.log(e);
}

ws.onopen = function (e) {
     console.log("OEPN");
     console.log(e);
}
```

The Handshake

- Upon a WebSocket object creation the browser sends an Upgrade request to the server

```
GET http://localhost:5481/socket/connect HTTP/1.1
Upgrade: websocket
Connection: Upgrade
Host: localhost:5481
Origin: http://localhost:5481
Sec-WebSocket-Key: rHeA9NhKxIuFX85mxpTOfQ==
Sec-WebSocket-Version: 13
```

- Server must respond with appropriate headers

Server Response


```
HTTP/1.1 101 Switching Protocols
Cache-Control: private
Upgrade: Websocket
Server: Microsoft-IIS/8.0
Sec-WebSocket-Accept: JLSwLlhPkGnb4qOJOnYzl957T7I=
Connection: Upgrade
```

- After a successful handshake, the data transfer part starts
- This is a two-way communication channel where each side can, independently from the other, send data at will

Send and Receive Messages

- WebSocket object supports
  - send – Can only be used after onopen event was fired
  - onmessage

```
var ws = new WebSocket("ws://localhost:5481/api/socket/connect");

ws.onclose = function (e) {
     ...
}

ws.onopen = function (e) {
    ws.send(message);
}

ws.onmessage = function (e) {
     console.log("MESSAGE: " + e.data);
}
```

Framing

- WebSocket is message based protocol
- The data being sent by the client is considered a message
- A message consists of multiple frames
- Each frame has slight overhead over the original payload
  - 2 bytes – FIN + Opcode + Payload Length + More
  - 4 bytes – Masking key
  - Above is true only for messages <= 125 bytes

Sub Protocol

- Client may specify a list of sub protocols
- Server must response with exactly one matched sub protocol
- WebSocket object contains a property named protocol which holds server selection

```
var ws = new WebSocket("ws://localhost:5481/api/socket/connect", ["myproto1", "myproto2"]);

ws.onopen = function (e) {
     console.log("OPEN: " + e.target.protocol);

     ws.send(message);
}
```

Summary

- Web Sockets brings “realtime-ness” to your web pages
- It is smarter and more efficient then just using polling
- However, you need both modern server and modern browser
- Consider using Web Socket Polyfills like SignalR

JSF and HTML5

- HTML 5 introduces many new elements and attributes.
- It would be a nightmare to maintain associate UIComponents for both HTML5 and previous versions.
- The solution is to use pass-through elements.

Pass-through Elements

- You use the regular HTML element (without the jsf prefix) and if on at least one attribute there is the prefix of (jsf: (http://xmlns.jcp.org/jsf)), it is treat as bound to the respective UIComponent.
- You can then use EL for all attributes.

- See full binding table:
- https://docs.oracle.com/javaee/7/tutorial/jsf-facelets009.htm

Pass-through Attributes

- JSF 2.2 added pass-through attributes which work the other way.
- If, on a JSF component the attribute is prefixed by (p: (http:// xmlns.jcp.org/jsf/passthrough)), then that attribute is ignored by JSF and pass to the browser.

Resources

- By default, JSF looks for resource in two locations:
  - The “resources” root directory.
  - “META-INF/resources” inside JARs.
- Note that in the first case, the directory is accessible from the outside.
- Since JSF 2.2, you can change the location of the first option by setting the context parameter *javax.faces.WEBAPP_RESOURCES_DIRECTORY* to a different location.

ResourceResolver

- Controls from where to load Facelets views.
- The default is the root of the application (not considering Flows).
- Before JSF 2.2, you could extend from this class and register your implementation under the context param: *javax.faces.FACELETS_RESOURCE_RESOLVER*.
- In your implementation you needed to return a URL according to a path given.

ResourceResolver

- In JSF 2.2, the *ResourceResolver* class has been deprecated.
- Its behavior has been generalized and merged into *ResourceHandler*.

- Let's discuss *ResourceHandler*…

ResourceHandler

- Before JSF 2.2, It was the base class used for implementing sending resources directly to the client (e.g., CSS, Scripts).
- Now it is also responsible for returning *ViewResources* (which supports a *getURL()*).

- Facelets was abstracted into a *ViewDefinitionLanguage* implementation (currently the only serious one).

Resource Library Contracts

- With JSF 2.2, it is possible to group resources (e.g., CSS) under a resource library contract.
- In the *faces-config.xml* you can define the contracts under the *application* tag.

- You basically map URL patterns (of the views) to a contract name.

Resource Library Contracts

- You need to create a “contracts” directory on the root folder and put your resources there.
- Then, views that access resources will automatically work within their contract.

- Remember to use <h:outputStylesheet name=…/> to insert the CSS link (otherwise no resolving will occur.

What is React?

- A declarative, efficient, and flexible JavaScript library for building user interfaces.
- One-way data flow
  - Properties, a set of immutable values, are passed to a component's renderer as properties in its HTML tag. A component cannot directly modify any properties passed to it, but can be passed callback functions that do modify values.
- Virtual DOM
  - React creates an in-memory data structure cache, computes the resulting differences, and then updates the browser's displayed DOM efficiently. This allows to write code as if the entire page is rendered on each change.
- JSX
  - React components are typically written in JSX, a JavaScript extension syntax allowing quoting of HTML and using HTML tag syntax to render subcomponents.

Environment Prerequisites

- Node.Js >= 6.10.3
- NPM package manager
- Webpack module bundler

Environment – Installing Webpack

- Webpack is a module bundler which takes modules with dependencies and generates static assets by bundling them together based on some configuration.
- The support of loaders in Webpack makes it a perfect fit for using it along with React and we will discuss it later in this post with more details.
- Webpack installation is done using NPM.

Environment – Installing Webpack

- Let's start by initiating an npm environment:

```
npm init
```

- Install required webpack using npm:

```
npm install webpack babel-loader babel-preset-es2015 babel-preset-react --save
```

- Now we have the required modules to create a working webpack bundler for a React environment.
- But, how do the pieces glue together? Check out the next few slides for the answer!

Environment – Installing Webpack

- Webpack requires a config file to be present in the root folder of your project run:

```
var webpack = require('webpack');
var path = require('path');

var BUILD_DIR = path.resolve(__dirname, 'build/');
var APP_DIR = path.resolve(__dirname, 'src/');

var config = {
    entry: APP_DIR + '/index.jsx',
    output: {
        path: BUILD_DIR,
        filename: 'bundle.js',
        module: {
            loaders: [
                {
                    test: /\.jsx?/,
                    include: APP_DIR,
                    loader: 'babel'
                }
            ]
        }
    }
};

module.exports = config;
```

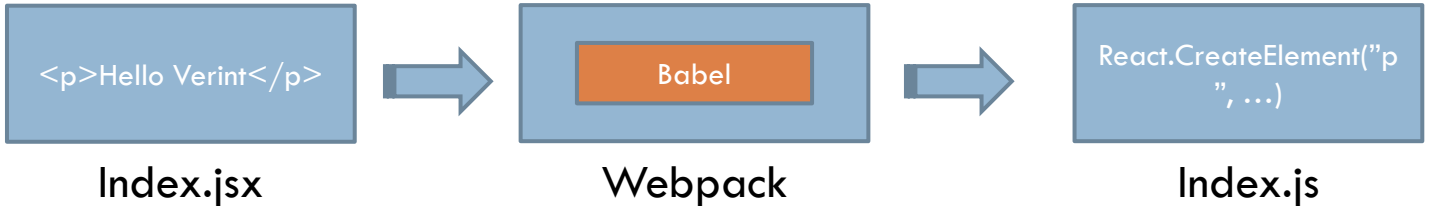
BUILD_DIR represents the directory path of the bundle file output.

The APP_DIR holds the directory path of the codebase

Tell's webpack which modules it should use in the build process

Environment – Setting up Babel

- JSX & ES6 syntax is not supported in most browsers by default. We need a tool which translates them to a format that is supported by the browsers. Babel is a tool that will do just that.



- Babel required a config file (.babelrc) to be present in the root folder of the project:

```
{
    "presets" : ["es2015", "react"]
}
```

Environment – Gluing the Pieces

- A React project that is built using webpack should look somewhat familiar to the following structure:

```
my-react-project/
├── .babelrc
├── package.json
├── webpack.config.js
├── src/
├── build/
```

- .babelrc
  - Babel transformer configuration.
- package.json
  - NPM configuration.
- webpack.config.js
  - Webpack build tool configuration.

Environment – Gluing the Pieces

- Create An index.html file in the build folder:

```
<html>
<head>
    <title>React Hello World</title>
</head>
<body>
    <div id="app"/>
    <script src="bundle.js" type="text/javascript"></script>
</body>
</html>
```

- Notice the div with the id “app”? It is the element which our app will hook to.
- bundle.js.. Eh? It is the output of the webpack build process. It contains the transformed code of the app.

Environment – Gluing the Pieces

- In-order to create a react app JSX entry point we need the react module to be present in our project:

```
npm install react react-dom --save
```

- Create an index.jsx file in the src folder:

```
import React from 'react';
import {render} from 'react-dom';

class HelloWorldApp extends React.Component {
    render() {
        return <p>Hello world!</p>
    }
}

render(HelloWorldApp, document.getElementById('app'));
```

- The following code represents a React component and a render function.

Environment – Gluing the Pieces

- After configuring the project, building is just a command away:

```
./node_modules/.bin/webpack -d --watch
```

- When the build process is done, the project's file structure should look somewhat similar to the following:

```
my-react-project/
├── .babelrc
├── package.json
├── webpack.config.js
├── src/
├── ├── index.jsx
├── build/
├── ├── index.html
├── ├── bundle.js
```

REACT COMPONENTS

Introduction

- Components let you split the UI into independent, reusable pieces, and think about each piece in isolation.
- Conceptually, components are like JavaScript functions. They accept arbitrary inputs (called "props") and return React elements describing what should appear on the screen.
- There are two main types of compoenets.
  - Stateful component – a component that hold's an internal state.
  - Pure component – a component that has no inner state but may receive data from it's props.

Creating a Component

- The simplest way to define a component is to write a JavaScript function:

```
const HelloWorld = () => (
    <div>Hello world!</div>
);
```

- This function is a valid React component because it accepts a single "props" object argument with data and returns a React element.
- You can also use an ES6 class to define a component:

```
class HelloWorld extends React.Component {
    render() {
        return (<div>Hello world!</div>);
    }
}
```

- The above two components are equivalent from React's point of view.

Rendering a Component

- When React sees an element representing a user-defined component, it passes JSX attributes to this component as a single object. We call this object "props".
- For example, this code renders "Hello, Itay" on the page:

```
const HelloWorld = (props) => (
    <div>Hello, {props.name}</div>
);

const element = <HelloWorld name="Itay"/>;

ReactDOM.render(element, document.body);
```

- But, what is JSX Really? Find out on the next few slides.

Lifecycle hooks

- componentWillMount()
  - Invoked immediately before mounting occurs. It is called before render().
- componentDidMount()
  - Invoked immediately after a component is mounted. Initialization that requires DOM nodes should go here.
- shouldComponentUpdate()
  - Use shouldComponentUpdate() to let React know if a component's output is not affected by the current change in state or props.
- componentWillUpdate()
  - Invoked immediately before rendering when new props or state are being received.
- componentDidUpdate()
  - Invoked immediately after updating occurs.
- componentWillReceiveProps()
  - Invoked before a mounted component receives new props.
- componentWillUnmount()
  - invoked immediately before a component is unmounted and destroyed.

Introduction to JSX

- JSX is a syntax extension to JavaScript. It is recommend to be used with React to describe what the UI should look like.
- JSX may remind you of a template language, but it comes with the full power of JavaScript. JSX produces React "elements".
- You can embed any JS expression in JSX by wrapping it in curly braces:

```
function sayHello(name) {
    return 'Hello, ' + name;
}

const name = 'Itay';

const element = (
    <h1>{sayHello(name)}</h1>
);

ReactDOM.render(element, document.body);
```

Introduction to JSX

- After compilation, JSX expressions become regular JavaScript objects. This means that you can use JSX inside of an if statements and for loops, assign it to variables, accept it as arguments, and return it from functions:

```
function GetGreeting(name) {
    if(name === 'Itay') {
        return <h1>Hello, Itay!</h1>;
    }
    return <h1>Hello, Stranger.</h1>;
}

const element = <GetGreeting name="Itay"/>;

ReactDOM.render(element, document.body);
```

Introduction to JSX

- JSX tags may contain children:

```
const element = (
    <div>
        <h1>Hello World!</h1>
        <p>How are you today?</p>
    </div>
);
```

- If a tag is empty, you may close it immediately with />:

```
const element = <img src={user.avatarUrl} />;
```

- Babel compiles JSX down to React.createElement() calls.
- The following examples are identical:

```
const element = (<h1 className="greeting">Hello, world!</h1>);

const element = React.createElement('h1', {className: 'greeting'}, 'Hello, world!');
```

JSX Pitfalls

- React Must Be in Scope:
  - JSX compiles into calls to React.createElement, the React library must also always be in scope.
- User-Defined Components Must Be Capitalized.
- Booleans, Null, and Undefined Are Ignored.
- There are several different ways to specify props in JSX:
  - You can pass any JavaScript expression as a prop, by surrounding it with {}.
  - You can pass a string literal as a prop.
  - If you pass no value for a prop, it defaults to true.
  - If you already have props as an object, and you want to pass it in JSX, you can use the spread operator to pass the whole props object.

Component – Props

- Whether you declare a component as a function or a class, it must never modify its own props. Consider this sum function:

```
function sum(a, b) {
    return a + b;
}
```

- Such functions are called pure because they do not attempt to change their inputs, and always return the same result for the same inputs.
- In contrast, this function is impure because it changes its own input:

```
function withdraw(account, amount) {
    account.total -= amount;
}
```

- React is pretty flexible but it has a single strict rule:
  - **All React components must act like pure functions with respect to their props.**

Component – Props Defaults

- You can setup default values for props which will be used on the component load. This code renders “Hello Itay”:

```
const HelloWorld = (props) => (
    <div>Hello, {props.name}</div>
);

HelloWorld.propTypes = {
    name: React.PropTypes.string
};

HelloWorld.defaultProps = {
    name: 'Itay'
};


const element = <HelloWorld />;

ReactDOM.render(element, document.body);
```

Component – State

- A component may or may not be stateful. The state is encapsulated inside a component and cannot be accessed directly from the outside but can be passed down to child components using props.

```
class Counter extends React.Component {
    constructor(props) {
        super(props);

        this.state = {
            counter: 0
        }
    }

    render() {
        return (
            <div>Counter: {this.state.counter}</div>
        );
    }
}

React.renderComponent(<Counter/>, document.body);
```

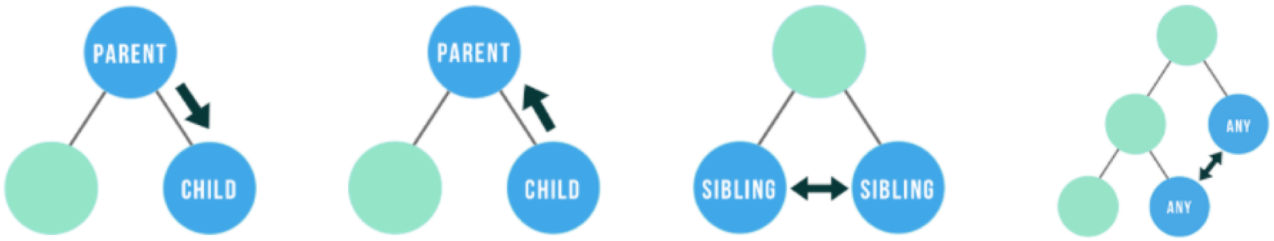
Component – Composing

- Components can refer to other components in their output. This lets us use the same component abstraction for any level of detail.
- If a component renders another component in it's render method, the renderer is the owner of the rendered component.
- The renderer owns the rendered component and has control over it. aka parenting.

```
const HelloWorldDisplay = (props) => (
    <div>Hello World!</div>
);

const App = () => (
    <div><HelloWorldDisplay/></div>
)

const element = <App />;

ReactDOM.render(element, document.body);
```

Interaction

- What if we had a button in a child component that needs to manipulates the state managed in a parent component?
- React component interaction comes in the form of data flow from **parent to child, child to parent and sibling to sibling.**
- Usually, component interacts with it's parent and siblings using callbacks (handlers) that are passed using props.



Parent to Child

1. Props
2. Instance Methods

Child to Parent

3. Callback Functions
4. Event Bubbling

Sibling to Sibling

5. Parent Component

Any to Any

6. Observer Pattern
7. Global Variables
8. Context

Interaction – Parent to Child

- Parent to child interaction is probably the simplest case as we just pass down props to the child:

```
const ChildComponent = (props) => (
    <div>
        <button>{props.buttonText}</button>
    </div>
);

class ParentComponent extends React.Component {
    render() {
        return (
            <div>
                <ChildComponent buttonText="Click Me!"/>
            </div>
        );
    }
}
```

Interaction – Child to Parent

- Child to parent interaction will again, in most cases happen over props. It is common to pass down callbacks through props:

```
const ChildComponent = (props) => (
    // Bind to the onClick event of button and invoke onClick callback when fired
    <div>
        <button onClick={props.onClick}>Click me</button>
    </div>
);

class ParentComponent extends React.Component {
    // Callback function that will be passed down to the child component
    handleChildClick(data) {
        console.log('Child component has returned with data:', data);
    }

    render() {
        // Pass down the callback function through the onClick prop
        return (
            <div>
                <ChildComponent onClick={this.handleChildClick}/>
            </div>
        );
    }
}
```

Interaction – Sibling to Sibling

- Sibling to sibling interaction is not very common. It requires us to keep a state in the parent component for each child component and pass down callbacks.
- The solution is complicated. I will present it hands-on.

Interaction – Any to Any

- It is an anti pattern.
- If you reach a state where you want all the component's in your app to communicate with each other, your are doing something very very wrong..

React Context (Experimental)

- Allows you to pass data down through the component hierarchy without the usage of props.
- It is experimental. React's documentation warn us about it:

> Context is an advanced and experimental feature. The API is likely to change in future releases.

- A context provider component is required. It should implement both childContextTypes and getChildContext:
  - *childContextTypes* - static property that allows you to declare the structure of the context object being passed to your component's descendants.
  - *getChildContext* - prototype method that returns the context object to pass down the component's hierarchy. Every time the component renders, this method will be called.

React Context (Experimental)

- The following code will show the interaction between a provider component and a consumer component:

```
class Provider extends React.Component {
    static childContextTypes = {
        versionId: React.PropTypes.string
    };

    getChildContext() {
        return {versionId: '1.0.0'};
    }

    render() {
        return (...);
    }
}

const Consumer = (props, context) => (
    <p>{context.versionId}</p>
);

ContextConsumer.contextTypes = {
    context: React.PropTypes.string
};
```

- Note that The consumer needed to declare what it want's from the context via *contextTypes*

Component Ref

- React supports a special attribute that you can attach to any component. The ref attribute takes a callback function, and the callback will be executed immediately after the component is mounted or unmounted.
- When the ref attribute is used on an HTML element, the ref callback receives the underlying DOM element as its argument.
- Use at your own risk. Try and accomplish most of your data flow needs using props.

```
class CustomTextInput extends React.Component {
    render() {
        // Use the `ref` callback to store a reference to the text input DOM
        // element in an instance field (for example, this.textInput).
        return (
            <div>
                <input type="text" ref={(input) => { this.myRef = input; }} />
            </div>
        );
    }
}
```

REACT VIRTUAL DOM

DOM

- DOM stands for *Document Object Model* and is an abstraction of a structured text. For web developers, this text is an HTML code. *Elements* of HTML become *nodes* in the DOM. So, while HTML is a text, the DOM is an in-memory representation of this text.
- The HTML DOM provides an interface (API) to traverse and modify the nodes:

```
const item = document.getElementById("li");
item.parentNode.removeChild(item);
```

DOM - Issues

- The HTML DOM is always tree-structured. Traversing trees is fairly easily. Unfortunately, easily doesn't mean quickly here.
- The DOM trees are huge nowadays as we push more and more towards dynamic web apps (SPA etc..) which may cause performance and maintenance issues.
- Consider the following: a DOM tree is made of thousands of elements with lots of methods that handle events - clicks, submits etc.. A typical jQuery-like event handler looks like this:
  - Find every DOM node that is interested in an event.
  - Update node if necessary.
- The following has two main issues, It's hard to manage and It's inefficient.

Virtual Dom

The paradigm was not invented by React but React uses it and provides it for free.

- PROS
  - Is an abstraction of the DOM.
  - Fast and efficient "diffing" algorithm.
  - Multiple frontends (JSX, hyperscript).
  - Detached from browser-specific implementations.
  - Can be used without React (i.e. as an independent engine).
- CONS
  - Full in-memory copy of the DOM (higher memory use).

Virtual Dom

- The virtual DOM is pretty similar to the “regular” DOM.
- In most cases, when you have an HTML code, it’s pretty straight forward to make it into a static React component:
- There are more, rather minor, differences between the DOMs:
  - key, ref and dangerouslySetInnerHTML do not exist in “real” DOM.
  - You can read about the differences on the React website.

ReactElement

- ReactElement is the primary type in React.
- ReactElement is light, stateless, immutable, virtual representation of a DOM Element.
- ReactElements live in the virtual DOM.
- Almost every HTML tag can be a ReactElement (div, p, etc..).
- Once defined, ReactElements can be rendered into the “real” DOM. The moment when React ceases to control the elements. They become slow, boring DOM nodes:

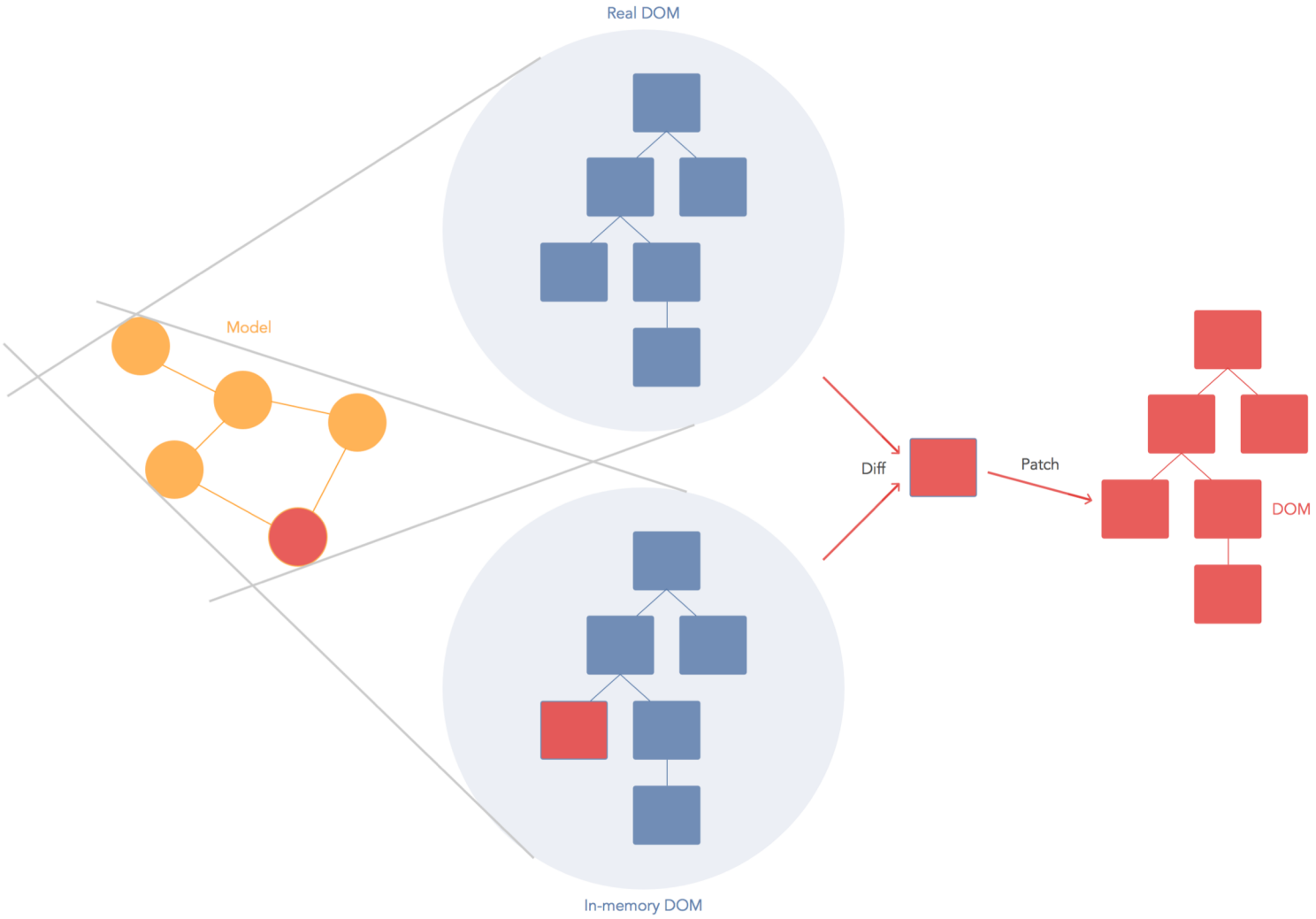
```
var root = React.createElement('div');
ReactDOM.render(root, document.getElementById('app'));
// JSX Equivalent
var root = <div />;
ReactDOM.render(root, document.getElementById('app'));
```

ReactComponent

- *ReactComponents are stateful.*
- ReactComponents don't have the access to the virtual DOM, but they can be easily converted to ReactElements:

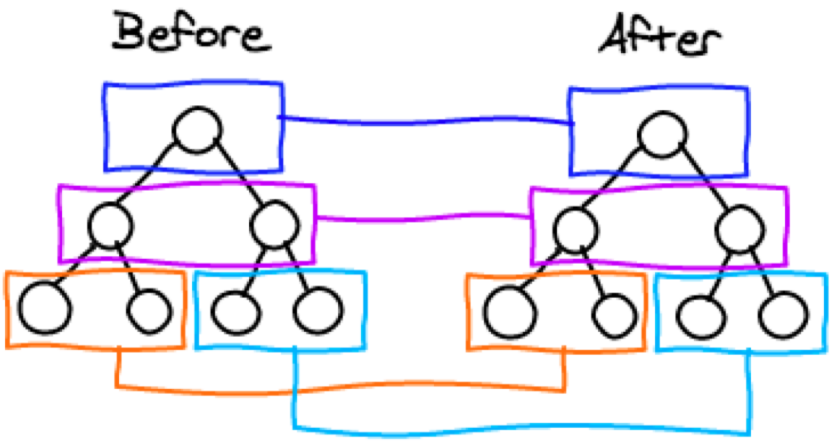
```
var element = React.createElement(MyComponent);
// or equivalently, with JSX
var element = <MyComponent />;
```

Reconciliation Process



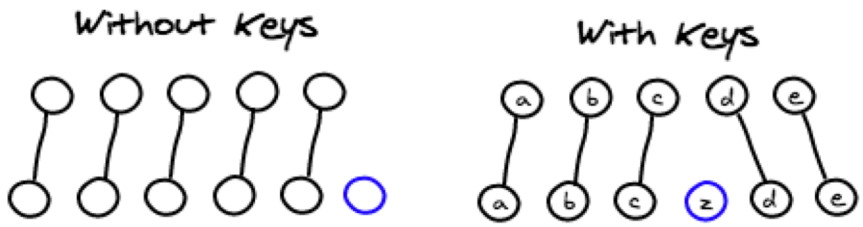
The Diff Algorithm – Level by level

- Finding the minimal number of modifications between two arbitrary trees is a O(n^3) problem. React uses simple and yet powerful heuristics to find a very good approximation in O(n).
- React reconcile trees level by level. This drastically reduces the complexity and isn't a big loss as it is very rare in web applications to have a component being moved to a different level in the tree. They usually only move laterally among children.



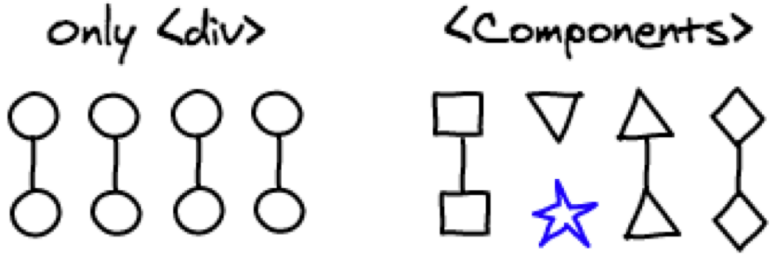
The Diff Algorithm – List

- Let say that we have a component that on one iteration renders 5 components and the next inserts a new component in the middle of the list. This would be really hard with just this information to know how to do the mapping between the two lists of components. By default, React associates the first component of the previous list with the first component of the next list, etc. You can provide a key attribute in order to help React figure out the mapping. In practice, this is usually easy to find out a unique key among the children.



The Diff Algorithm – Components

- A React app is usually composed of many user defined components that eventually turns into a tree composed mainly of divs. This additional information is being taken into account by the diff algorithm as React will match only components with the same class. For example, if a is replaced by an <fooBlock>, React will remove the div and create a foo block. We don't need to spend precious time trying to match two components that are unlikely to have any resemblance.

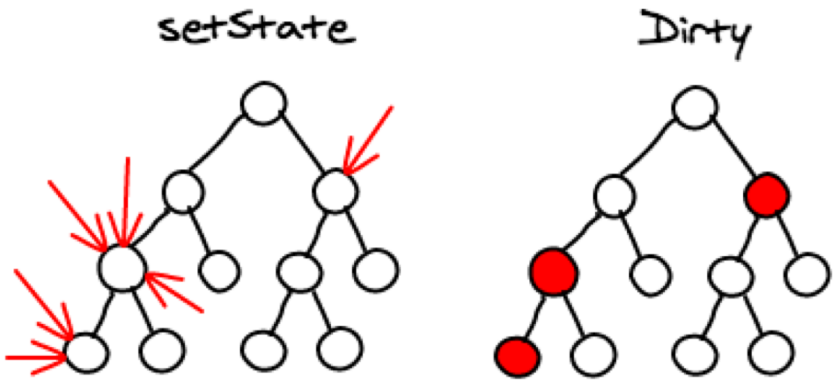


Reconcile - Event Delegation

- Attaching event listeners to DOM nodes is painfully slow and memory-consuming. Instead, React implements a popular technique called “event delegation”. React goes even further and re-implements a W3C compliant event system.
- it’s implemented as follows: A single event listener is attached to the root of the document. When an event is fired, the browser gives us the target DOM node. In order to propagate the event through the DOM hierarchy, React doesn’t iterate on the virtual DOM hierarchy. Instead, we use the fact that every React component has a unique id that encodes the hierarchy. We can use simple string manipulation to get the id of all the parents. By storing the event listeners in a hash map, we found that it performed better than attaching them to the virtual DOM.

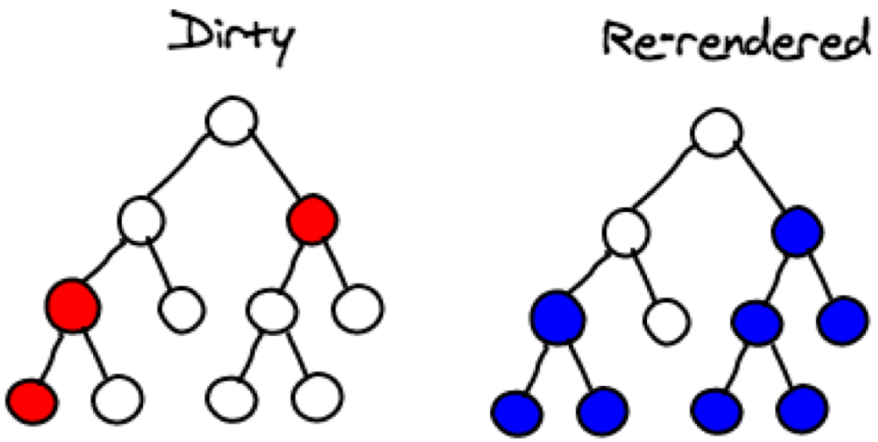
Reconcile - Rendering

- Whenever you set a state on a component, React will mark it as dirty. At the end of the event loop, React looks at all the dirty components and re-renders them. This batching means that during an event loop, there is exactly one time when the DOM is being updated. This property is key to building a performant app and yet is extremely difficult to obtain using commonly written JavaScript. In a React application, you get it by default.



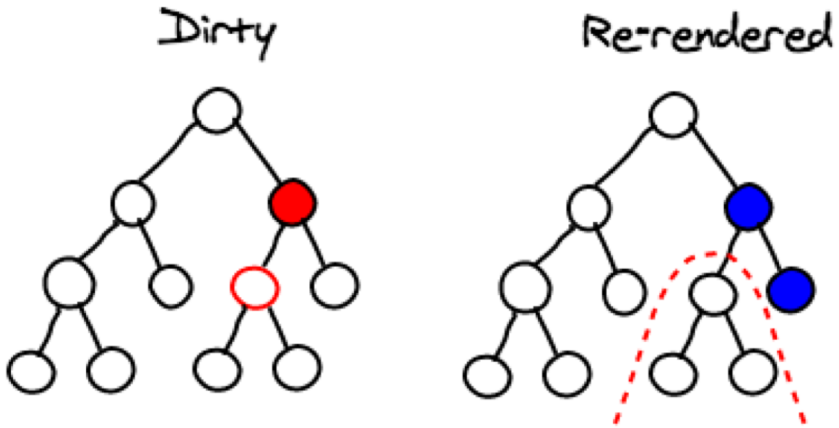
Reconcile - Sub-tree Rendering

- When setState function is called, the component rebuilds the virtual DOM for its children. If you call setState on the root element, then the entire React app is re-rendered. All the components, even if they didn't change, will have their render method called. This may sound scary and inefficient but in practice, this works fine because we're not touching the actual DOM.



Reconcile - Selective Sub-tree Rendering

- It is possible to prevent some sub-trees to re-render. If you implement the shouldComponentUpdate(nextProps, nextState) method on a component.
- Keep in mind that this function is going to be called all the time, so you want to make sure that it takes less time to compute than heuristic than the time it would have taken to render the component, even if re-rendering was not strictly needed.



Conclusion

- The techniques that make React fast are not new.
- We want to make as little changes to the “real” DOM.
- In practice, DOM optimizations are very hard to implement in regular JavaScript code. React gives you the optimizations ”On the house”.
- Simple performance cost model: every setState re-renders the whole sub-tree. If you want to squeeze out performance, call setState as low as possible and use shouldComponentUpdate to prevent re-rendering an large sub-tree.