Chapter 94: Web Cryptography API

Section 94.1: Creating digests (e.g. SHA-256)

```
// Convert string to ArrayBuffer. This step is only necessary if you wish to hash a string, not if
you already got an ArrayBuffer such as an Uint8Array.
var input = new TextEncoder('utf-8').encode('Hello world!');

// Calculate the SHA-256 digest
crypto.subtle.digest('SHA-256', input)
// Wait for completion
.then(function(digest) {
  // digest is an ArrayBuffer. There are multiple ways to proceed.

  // If you want to display the digest as a hexadecimal string, this will work:
  var view = new DataView(digest);
  var hexstr = '';
  for(var i = 0; i < view.byteLength; i++) {
    var b = view.getUint8(i);
    hexstr += '0123456789abcdef'[(b & 0xf0) >> 4];
    hexstr += '0123456789abcdef'[(b & 0x0f)];
  }
  console.log(hexstr);

  // Otherwise, you can simply create an Uint8Array from the buffer:
  var digestAsArray = new Uint8Array(digest);
  console.log(digestAsArray);
})
// Catch errors
.catch(function(err) {
  console.error(err);
});
```

The current draft suggests to provide at least `SHA-1`, `SHA-256`, `SHA-384` and `SHA-512`, but this is no strict requirement and subject to change. However, the SHA family can still be considered a good choice as it will likely be supported in all major browsers.

Section 94.2: Cryptographically random data

```
// Create an array with a fixed size and type.
var array = new Uint8Array(5);

// Generate cryptographically random values
crypto.getRandomValues(array);

// Print the array to the console
console.log(array);
```

`crypto.getRandomValues(array)` can be used with instances of the following classes (described further in Binary Data) and will generate values from the given ranges (both ends inclusive):

- `Int8Array`: -27 to 27-1
- `Uint8Array`: 0 to 28-1
- `Int16Array`: -215 to 215-1
- `Uint16Array`: 0 to 216-1
- `Int32Array`: -231 to 231-1
- `Uint32Array`: 0 to 231-1

Section 94.3: Generating RSA key pair and converting to PEM format

In this example you will learn how to generate RSA-OAEP key pair and how to convert private key from this key pair to base64 so you can use it with OpenSSL etc. Please note that this process can also be used for public key you just have to use prefix and suffix below:

```
-----BEGIN PUBLIC KEY-----
-----END PUBLIC KEY-----
```

NOTE: This example is fully tested in these browsers: Chrome, Firefox, Opera, Vivaldi

```
function arrayBufferToBase64(arrayBuffer) {
    var byteArray = new Uint8Array(arrayBuffer);
    var byteString = '';
    for(var i=0; i < byteArray.byteLength; i++) {
        byteString += String.fromCharCode(byteArray[i]);
    }
    var b64 = window.btoa(byteString);

    return b64;
}

function addNewLines(str) {
    var finalString = '';
    while(str.length > 0) {
        finalString += str.substring(0, 64) + '\n';
        str = str.substring(64);
    }

    return finalString;
}

function toPem(privateKey) {
    var b64 = addNewLines(arrayBufferToBase64(privateKey));
    var pem = "-----BEGIN PRIVATE KEY-----\n" + b64 + "-----END PRIVATE KEY-----";

    return pem;
}

// Let's generate the key pair first
window.crypto.subtle.generateKey(
    {
        name: "RSA-OAEP",
        modulusLength: 2048, // can be 1024, 2048 or 4096
        publicExponent: new Uint8Array([0x01, 0x00, 0x01]),
        hash: {name: "SHA-256"} // or SHA-512
    },
    true,
    ["encrypt", "decrypt"]
).then(function(keyPair) {
    /* now when the key pair is generated we are going
       to export it from the keypair object in pkcs8
    */
    window.crypto.subtle.exportKey(
        "pkcs8",
        keyPair.privateKey
    ).then(function(exportedPrivateKey) {
        // converting exported private key to PEM format
        var pem = toPem(exportedPrivateKey);
        console.log(pem);
    }).catch(function(err) {
        console.log(err);
    });
});
```

That's it! Now you have a fully working and compatible RSA-OAEP Private Key in PEM format which you can use wherever you want. Enjoy!

Section 94.4: Converting PEM key pair to CryptoKey

So, have you ever wondered how to use your PEM RSA key pair that was generated by OpenSSL in Web Cryptography API? If the answers is yes. Great! You are going to find out.

NOTE: This process can also be used for public key, you only need to change prefix and suffix to:

```
-----BEGIN PUBLIC KEY-----
-----END PUBLIC KEY-----
```

This example assumes that you have your RSA key pair generated in PEM.

```
function removeLines(str) {
    return str.replace("\n", "");
}

function base64ToArrayBuffer(b64) {
    var byteString = window.atob(b64);
    var byteArray = new Uint8Array(byteString.length);
    for(var i=0; i < byteString.length; i++) {
        byteArray[i] = byteString.charCodeAt(i);
    }

    return byteArray;
}

function pemToArrayBuffer(pem) {
    var b64Lines = removeLines(pem);
    var b64Prefix = b64Lines.replace('-----BEGIN PRIVATE KEY-----', '');
    var b64Final = b64Prefix.replace('-----END PRIVATE KEY-----', '');

    return base64ToArrayBuffer(b64Final);
}

window.crypto.subtle.importKey(
    "pkcs8",
    pemToArrayBuffer(yourprivatekey),
    {
        name: "RSA-OAEP",
        hash: {name: "SHA-256"} // or SHA-512
    },
    true,
    ["decrypt"]
).then(function(importedPrivateKey) {
    console.log(importedPrivateKey);
}).catch(function(err) {
    console.log(err);
});
```

And now you're done! You can use your imported key in WebCrypto API.