Medii de proiectare și programare

2017-2018
Curs 10

Conținut

- Servicii REST - Clienți web
- JavaScript: Promise, Fetch
- React JS

JavaScript - Evenimente

- JavaScript este secvențial (eng. *single threaded*), două părți de cod nu pot fi executate în același timp, trebuie să fie executate una după cealaltă.
- În browsere, JavaScript împarte un fir de execuție cu alte sarcini, care diferă de la browser la browser.
- JavaScript se comportă similar cu firul de execuție corespunzător interfețelor grafice: orice modificare a interfeței grafice duce la amânarea următoarele sarcini corespunzătoare ei.
- În JavaScript se pot folosi evenimente și funcții callback:

```
var img1 = document.querySelector('.img-1');

img1.addEventListener('load', function() {
  // imaginea a fost încărcată
});

img1.addEventListener('error', function() {
  //  a apărut o situație excepțională
});
```

JavaScript - Evenimente

- Este posibil ca un eveniment să apară înainte de a adăuga un listener. (Se poate folosi proprietarea "*complete*" pentru a trata situația):

```
var img1 = document.querySelector('.img-1');

function loaded() {
  // imaginea a fost încărcată

}

if (img1.complete) {
  loaded();
}
else {
  img1.addEventListener('load', loaded);
}

img1.addEventListener('error', function() {
  // a apărut o situație excepțională
});
```

- Această soluție nu tratează cazul apariției unei erori înaintea adăugării listenerului.
- Soluția devine complexă când dorim să tratăm cazul încărcării mai multor imagini.
- Evenimentele sunt utile pentru situații care apar de mai multe ori asupra aceluiași obiect (apăsarea unei taste, etc). In aceste cazuri nu ne interesează ce s-a întâmplat înaintea adăugării listenerului.

JavaScript - Promise

- Pentru tratarea rezultatului unei operații asincrone (succes/eșec), este preferat următorul șablon:

```
img1.callThisIfLoadedOrWhenLoaded(function() {
  // încărcată
}).orIfFailedCallThis(function() {
  // eșec
});

// și…
whenAllTheseHaveLoaded([img1, img2]).callThis(function() {
  // toate imaginile au fost încărcate
}).orIfSomeFailedCallThis(function() {
  // încărcarea unei imagini sau a mai multor imagini a eșuat
});
```

- Conceptul de promise a fost introdus pentru astfel de situații.
- Promises sunt asemănătoare cu listeneri, exceptând:
    - Rezultatul execuției unui promise poate fi succes/eșec o singură dată. Nu poate fi semnalat succesul/eșecul de mai multe ori. Nu poate fi schimbat rezultatul succes -> eșec și invers.
    - Dacă rezultatul execuției este succes/eșec, dar funcția callback este adăugată ulterior, funcția va fi apelată cu rezultatul corect, chiar dacă rezultatul s-a obținut anterior.
- Este utila obținerea rezultatului unei operații asincrone (succes/eșec), deoarece este mai important rezultatul obținut decât momentul de timp în care a fost obținut.

JavaScript - Promise

- Un obiect *Promise* reprezintă eventualul rezultat al unei operații asincrone. Modalitatea principală de a interacționa cu un promise este de a adăuga funcții callback care vor fi apelate fie cu rezultatul execuției (succes), fie cu motivul eșecului.
- Un *Promise* reprezintă o valoare care poate fi disponibilă acum sau în viitor sau niciodată. Valoarea respectivă e posibil să nu fie cunoscută în momentul creării obiectului.
- Permite adăugarea handlerelor (funcțiile callback) pentru tratarea succesului/eșecului.
- Permite metodelor asincrone să returneze un rezultat asemănător cu metodele sincrone: în loc să returneze rezultatul imediat, metoda asincronă returnează un obiect Promise cu ajutorul căreia poate fi obținut rezultatul execuției.
- Concepte:
  - *Promise*: un obiect cu o metodă *then,* a cărui execuției este conformă cu specificația.
  - *Thenable:* un obiect care definește o metodă *then.*
  - *Valoare:* orice valoare permisă în JavaScript (inclusiv *undefined,* un *thenable* sau un *promise*)
  - *Excepție*: o valoare aruncată folosind *throw.*
  - *Motiv:* o valoare care indică motivul respingerii unui promise.

JavaScript - Promise

```
new Promise( /* executor */ function(resolve, reject) { ... } );
```

- *executor:* O funcție cu doi parametri *resolve* și *reject (*funcții callback).
  - Funcția *executor* este executată imediat de implementarea Promise, transmițând funcțiile *resolve* și *reject* (Executorul este apelat înainte de ieșirea din constructorul corespunzător obiectului promise).
  - Funcțiile *resolve/reject* când sunt apelate acceptă/resping promise-ul.
    - De obicei, executorul inițiază cod asincron, iar după încheierea execuției apelează fie funcția *resolve*(succes) sau *reject* (eșec, eroare).
- Dacă o eroare este aruncată în timpul execuției executorului, promise-ul este respins (rejected). Valoarea returnată de executor în acest caz este ignorată.

JavaScript - Stările Promise

- Un obiect *Promise* poate fi în una din următoarele stări:
  - *pending:* stare inițială, încă nu a fost îndeplinit sau respins (operația asincronă încă se execută).
  - *fulfilled:* execuția operației asincrone s-a încheiat cu succes (îndeplinit).
  - *rejected:* execuția operației asincrone a eșuat (eroare) (respins).
- Un promise în starea *pending* poate fi îndeplinit cu o anumită valoare, sau poate fi respins pe baza unui motiv (eroarea).
- În oricare dintre cele două situații, handlerele asociate folosind metoda *then* sunt apelate.
- Dacă un promise a fost deja îndeplinit sau respins în momentul atașării unui handler, handler va fi apelat cu rezultatul execuției promise-ului.
- Un promise este *stabilit (eng. settled)* dacă a fost fie îndeplinit, fie respins, dar nu este în starea de pending.
- Se mai folosește termenul rezolvat (eng. *rezolved*) - înseamnă că obiectul promise este stabilit, sau este într-o înlănțuire de obiecte promise.

JavaScript - Promise

- Crearea unui obiect Promise:

```
var promise = new Promise(function(resolve, reject) {
  // codul (asincron)

  if (/* totul s-a executat cu succes */) {
    resolve("Succes!");
  }
  else {
    reject(Error("Eroare"));
  }
});
```

- Constructorul primește un singur parametru, un executor. La finalul execuției codului (asincron) se apelează fie funcția *resolve*, fie funcția *reject*.
- Se recomandă respingerea unui promise folosind un obiect de tip Error (posibilitatea obținerii stivei de execuției, depanare mai ușoară).

JavaScript - Promises

- Folosirea unui promise:

```
promise.then(function(result) {
  console.log(result); // "Succes!"
}, function(err) {
  console.log(err); // Error: "Eroare"
});
```

- Funcția *then()* primește doi parametri, un callback pentru execuția cu succes, și un callbak pentru eșec. Ambele funcții sunt opționale, se poate adăuga doar un callback pentru succes sau pentru eșec.
- JavaScript promises pot fi executate și în afara browserelor (ex. folosind NodeJS).
- DOM folosește promises. Funcțiile asincrone din noul DOM APIs folosesc promises (ex. ServiceWorker, Streams, etc.).

Înlănțuirea Promise

- Obiectele promise pot fi înlănțuite, pentru a transforma valorile obținute sau pentru a executa asincron alt cod, unul după altul.

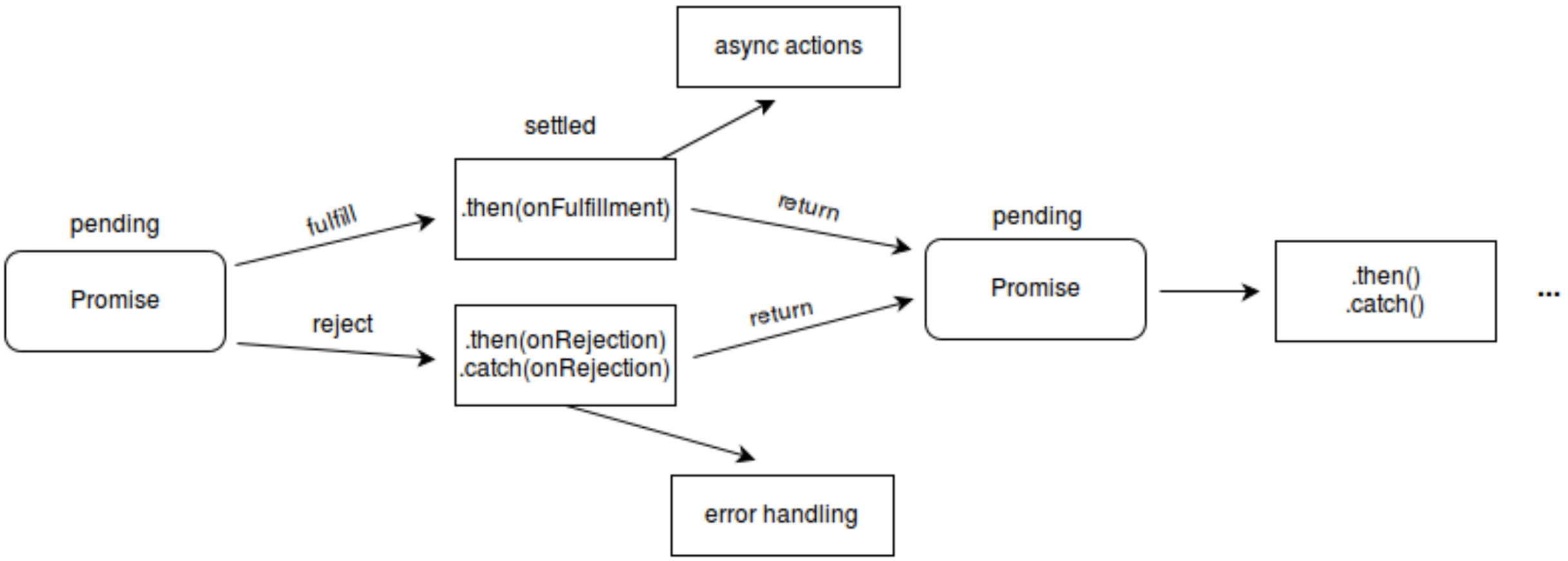
```
var promise = new Promise(function(resolve, reject) {
  resolve(1);
});

promise.then(function(val) {
  console.log(val); // 1
  return val + 2;
}).then(function(val) {
  console.log(val); // 3
})

// ———

get('story.json').then(function(response) {
  console.log("Success!", response);   //response e in format text
})
```

Înlănțuire Promises (*chaining*)



JavaScript - Promises chaining

```
get('story.json').then(function(response) {
  return JSON.parse(response);
}).then(function(response) {
  console.log(“Format JSON!", response);
})
```

- *JSON.parse()* primește un singur parametru și returnează o valoare transformată, se poate folosi și formatul:

```
get('story.json').then(JSON.parse).then(function(response) {
  console.log(“Format JSON:”, response);
})

function getJSON(url) {
  return get(url).then(JSON.parse);
}
```

- Funcția *getJSON()* returnează un alt promise, care conține obiectul JSON obținut după parsarea răspunsului.

Promises - Înlănțuirea acțiunilor asincrone

- Înlănțuirea folosind funcția *then* poate fi folosită și pentru a executa secvențial mai multe operații asincrone (contează ordinea executării lor).
- Dacă metoda *then()* returnează o valoare, următorul apel al funcției *then* primește valoarea respectivă ca și parametru.
- Dacă returnează un alt obiect promise, următorul apel al funcției *then* așteaptă terminarea execuției acestuia, și va fi apelat când obiectul devine *settled* (succes/eșec).

```
getJSON('story.json').then(function(story) {
  return getJSON(story.chapterUrls[0]);
}).then(function(chapter1) {
  console.log("Got chapter 1!", chapter1);
})
```

Promises -Tratarea excepțiilor

- Funcția *then()* primește doi parametri: callback succes și callback eșec:

```
get('story.json').then(function(response) {
  console.log("Success!", response);
}, function(error) {
  console.log("Failed!", error);
})
```

- Se poate folosi și funcția *catch()*:

```
get('story.json').then(function(response) {
  console.log("Success!", response);
}).catch(function(error) {
  console.log("Failed!", error);
})
```

- *Catch()* este sinonim cu *then(undefined, func)*, dar este mai ușor de înțeles.

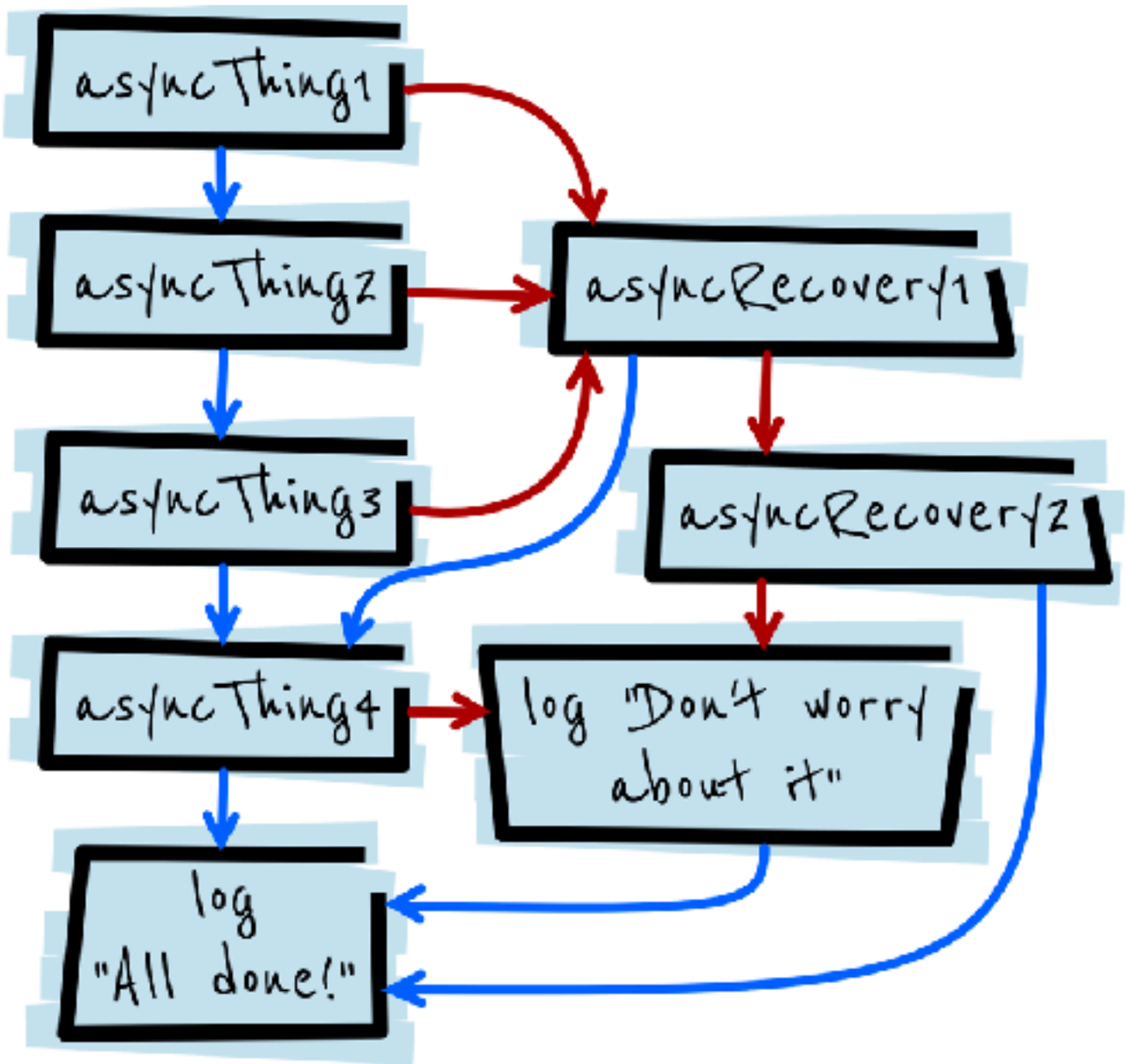
- Cele două exemple de cod, nu sunt echivalente (ultimul este echivalent cu):

```
get('story.json').then(function(response) {
  console.log("Success!", response);
}).then(undefined, function(error) {
  console.log("Failed!", error);
})
```

Promises - Tratarea excepțiilor

- Dacă un promise este respins (rejected), execuția se mută la următorul apel *then* care conține un callback pentru cazul respingerii, sau până la întâlnirea unui apel *catch.*
- În situația *then(func1, func2),* se apelează *func1* sau func2 dar nu amândouă.
- În situația *then(func1).catch(func2)*, amândouă funcțiile vor fi apelate dacă func1 este respinsă (fac parte din obiecte promise diferite)

```
asyncThing1().then(function() {
  return asyncThing2();
}).then(function() {
  return asyncThing3();
}).catch(function(err) {
  return asyncRecovery1();
}).then(function() {
  return asyncThing4();
}, function(err) {
  return asyncRecovery2();
}).catch(function(err) {
  console.log("Don't worry about it");
}).then(function() {
  console.log("All done!");
})
```



JavaScript excepții și promises

- Respingerea are loc când un obiect promise este respins explicit (prin apelul callbackului *reject*), dar și când o eroare este aruncată în executorul obiectului promise:

```
var jsonPromise = new Promise(function(resolve, reject) {
  // JSON.parse aruncă eroare dacă stringul nu este în format json
  //  JSON invalid, obiectul promise este respins (rejected) implicit:
  resolve(JSON.parse("Nu e format JSON"));
});
```

```
jsonPromise.then(function(data) {
  // NU va fi apelat niciodată:
  console.log("Succes!", data);
}).catch(function(err) {
  // Va fi apelat:
  console.log("Esec!", err);
})
```

- Se recomandă execuția codului asincron în executor, astfel erorile vor fi prinse și vor genera automat respingerea.

JavaScript excepții și promises

- Aceeași situație pentru erorile aruncate în funcțiile callback transmise funcției then():

```
get('/').then(JSON.parse)
 .then(function() {
  // Eroare, '/' este o pagina HTML, nu este in format JSON
  //  JSON.parse arunca exceptie
  console.log("Succes!", data);
}).catch(function(err) {
  // Se va executa:
  console.log("Esec!", err);
})
```

JavaScript Promises

- Funcția *Promise.resolve(val)*, creează un promise care se va îndeplini cu valoarea transmisă ca și parametru.
  - Dacă parametrul este un alt promise, acesta va fi returnat.
  - Exemplu: Promise.resolve(‘Ana') creează un obiect promise, care va fi îndeplinit cu valoarea ‘Ana’.
  - Dacă este apelat fără parametri, va fi îndeplinit cu valoarea "*undefined*".
- Funcția *Promise.reject(val)*, creează un promise care va fi respins cu valoarea transmisă ca și parametru (sau *undefined*).
- Funcția *Promise.all* primește un tablou de obiecte promise și creează un obiect promise care va fi îndeplinit dacă toate sunt îndeplinite. Valoarea este un tablou cu rezultatele obținute la îndeplinirea obiectelor promise, în ordinea acestora din tabloul inițial.

```
Promise.all(arrayOfPromises).then(function(arrayOfResults) {
  //...
})
```

JavaScript Promises

Constructor	
`new Promise(function(resolve, reject) {});`	**`resolve(thenable)`** Your promise will be fulfilled/rejected with the outcome of **`thenable`** **`resolve(obj)`** Your promise is fulfilled with **`obj`** **`reject(obj)`** Your promise is rejected with **`obj`**. For consistency and debugging (e.g., stack traces), obj should be an **`instanceof Error`**. Any errors thrown in the constructor callback will be implicitly passed to **`reject()`**.

Instance Methods	
`promise.then(onFulfilled, onRejected)`	**`onFulfilled`** is called when/if "promise" resolves. **`onRejected`** is called when/if "promise" rejects. Both are optional, if either/both are omitted the next **`onFulfilled/onRejected`** in the chain is called. Both callbacks have a single parameter, the fulfillment value or rejection reason. **`then()`** returns a new promise equivalent to the value you return from **`onFulfilled/onRejected`** after being passed through **`Promise.resolve`**. If an error is thrown in the callback, the returned promise rejects with that error.
`promise.catch(onRejected)`	Sugar for **`promise.then(undefined, onRejected)`**

JavaScript Promises

- Metode statice

Method summaries	
`Promise.resolve(promise);`	Returns promise (only if **`promise.constructor == Promise`**)
`Promise.resolve(thenable);`	Make a new promise from the thenable. A thenable is promise-like in as far as it has a `then()` method.
`Promise.resolve(obj);`	Make a promise that fulfills to **`obj`**. in this situation.
`Promise.reject(obj);`	Make a promise that rejects to **`obj`**. For consistency and debugging (e.g. stack traces), **`obj`** should be an **`instanceof Error`**.
`Promise.all(array);`	Make a promise that fulfills when every item in the array fulfills, and rejects if (and when) any item rejects. Each array item is passed to **`Promise.resolve`**, so the array can be a mixture of promise-like objects and other objects. The fulfillment value is an array (in order) of fulfillment values. The rejection value is the first rejection value.
`Promise.race(array);`	Make a Promise that fulfills as soon as any item fulfills, or rejects as soon as any item rejects, whichever happens first.

JavaScript -Fetch

- Funcția fetch() permite executarea unor apeluri prin rețea asemănătoare cu XMLHttpRequest (XHR).
- Diferența: Fetch API folosește obiecte promise, care permit scrierea unui cod mai simplu și mai clar, evitând callbackurile complexe și API complex al XMLHttpRequest.
- Exemplu XMLHttpRequest: cererea unui URL, obținerea unui răspuns și parsarea lui ca și JSON.

```
function reqListener() {
  var data = JSON.parse(this.responseText);
  console.log(data);
}

function reqError(err) {
  console.log('Eroare:', err);
}

var oReq = new XMLHttpRequest();
oReq.onload = reqListener;
oReq.onerror = reqError;
oReq.open('get', './api/some.json', true);
oReq.send();
```

JavaScript -Fetch

- Solutia fetch()

```
fetch('./api/some.json')
  .then(
    function(response) {
      if (response.status !== 200) {
        console.log('Eroare. Status Code: ' +  response.status);
        return;
      }

      // Examinarea textului din răspuns
      response.json().then(function(data) {
        console.log(data);
      });
    }
  )
  .catch(function(err) {
    console.log(‘Eroare Fetch ', err);
  });
```

- Rezultatul unui apel *fetch()* este un obiect de tip Stream. Se poate apela metoda *json()*, iar rezultatul va fi un obiect de tip Promise, deoarece citirea streamului se va face asincron.

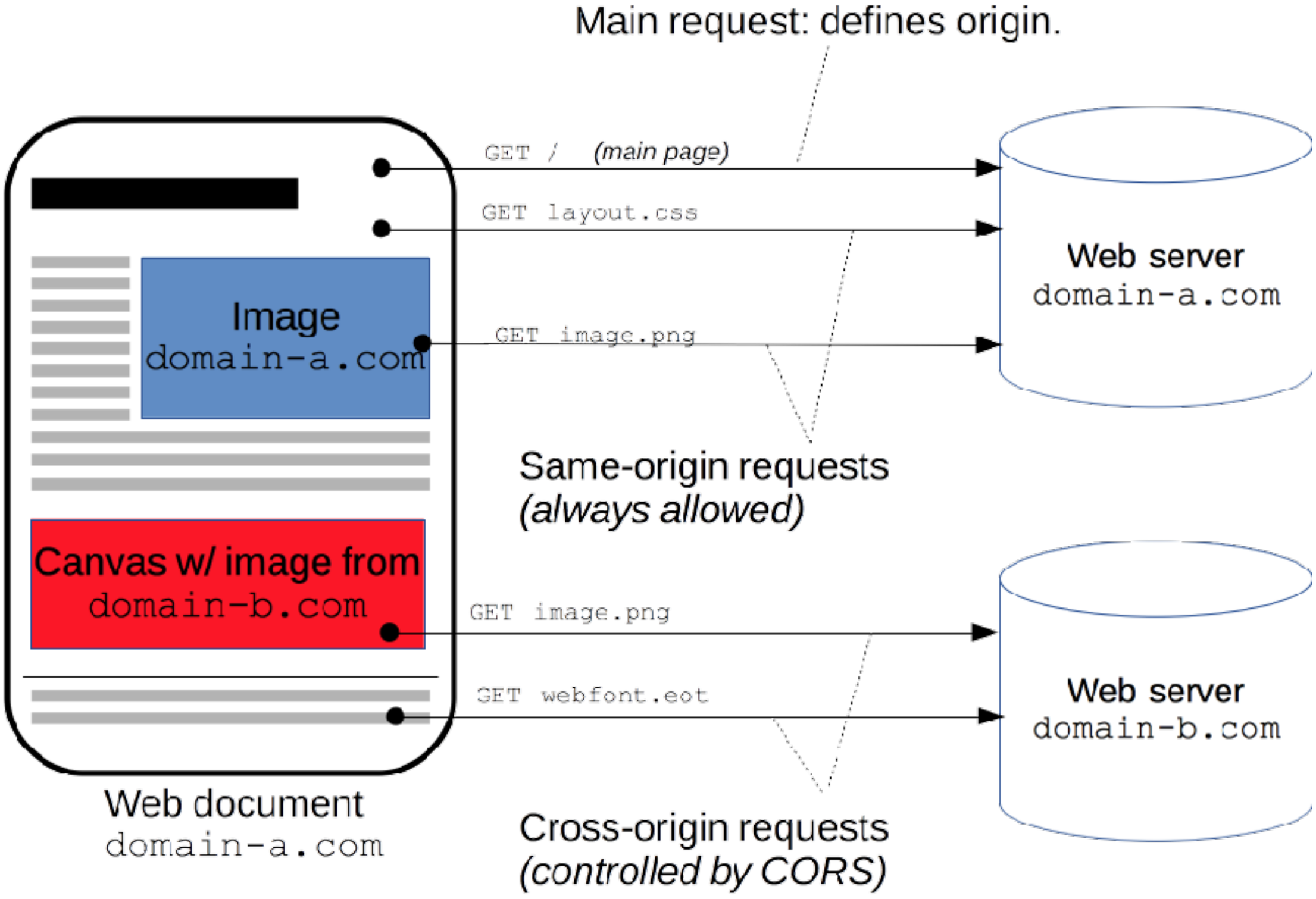
Fetch Response

- Apelul funcției *fetch* returnează obiecte de tip response, dacă obiectul de tip promise asociat este îndeplinit.
- Proprietăți utile:
  - Response.status - status code asociat răspunsului (întreg, implicit 200).
  - Response.statusText - textul asociat status code-ului asociat (string, implicit “OK”).
  - Response.ok - permite verificarea rapida daca status-ul este între 200-299 (boolean).
- Alte informații care pot fi obținute (ex. antetele):

```
fetch('users.json').then(function(response) {
    console.log(response.headers.get('Content-Type'));
    console.log(response.headers.get('Date'));

    console.log(response.status);
    console.log(response.statusText);
    console.log(response.type);
    console.log(response.url);
});
```

Cross Origin Resource Sharing (CORS)



https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

Fetch Response Types

- Când se face o cerere fetch, răspunsul va primi un tip (*response.type*) de tip "*basic*", "*cors*" sau "*opaque*". Aceste tipuri indică de unde provine resursa și poate fi folosit pentru a decide modalitatea de tratare a obiectului de tip response.
    - *basic*: se face o cerere pentru o resursă având aceeași origine ca și cererea. Nu există restricții asupra informațiilor ce pot fi obținute din răspuns.
    - *cors*: se face o cerere pentru o resursă a cărei origine este diferită și care returnează *Cross Origin Resource Sharing* (CORs) în antet. Un răspuns de tip *cors* restricționează antetele care pot fi accesate la `Cache-Control`, `Content-Language`, `Content-Type`, `Expires`, `Last-Modified`, și `Pragma`.
- *opaque*: se face o cerere pentru o resursă având altă origine și care nu returnează CORS în antet. La acest tip de răspuns nu se pot obține datele returnate sau status-ul răspunsului.
    - Nu se poate verifica dacă cererea s-a efectuat cu succes.

Fetch Response Types

- Se poate defini un mod pentru o cerere fetch, astfel încât doar anumite cereri vor fi apelate:
  - *same-origin:* se execută cu succes doar pentru cereri având aceeași origine, alte cereri vor fi respinse.
  - *cors:* permite cereri având aceeași origine, sau către alte origini care conțin antetele CORs corespunzătoare.
  - *cors-with-forced-preflight:* se verifică anterior că cererea poate fi făcută.
  - *no-cors:* pentru cereri către alte origini care nu au setat antetul CORS, răspunsul va fi de tip opaque.
- Pentru definirea modului se adaugă un parametru cererii fetch:

```
fetch('http://some-site.com/cors-enabled/some.json', {mode: 'cors'})
  .then(function(response) {
    return response.text();
  })
  .then(function(text) {
    console.log('Request successful', text);
  })
  .catch(function(error) {
    log('Request failed', error)
  });
```

Fetch - înlănțuirea promise

- Pași cerere HTTP:
  - verificarea statusului răspunsului
  - parsarea conținutului pentru a obține obiectul JSON.
- Soluție care necesită doar folosirea informației obținute.

```
function status(response) {
  if (response.status >= 200 && response.status < 300) {
    return Promise.resolve(response)
  } else {
    return Promise.reject(new Error(response.statusText))
  }
}

function json(response) {
  return response.json()
}

fetch('users.json')
  .then(status)
  .then(json)
  .then(function(data) {
    console.log(‘Succes -  raspuns JSON:’, data);
  }).catch(function(error) {
    console.log(‘Cerere esuata', error);
  });
```

Fetch - Verificarea răspunsului

- Un obiect promise *fetch()* va respinge cu un obiect de tip TypeError, când apare o eroare de rețea.
- Un apel al funcției *fetch* ar trebui să verifice execuția cu succes a acesteia, verificând că obiectul de tip promise a fost îndeplinit, iar apoi verificând valoarea proprietății Response.ok.
- Exemplu:

```
fetch('some.json').then(function(response) {
  if(response.ok) {
    return response.json();
  }
  throw new Error('Raspunsul nu a fost ok.');
}).then(function(data) {
 //folosirea informatiei
}).catch(function(error) {
  console.log('Eroare cu functia fetch: ' + error.message);
});
```

Fetch - Request

- Funcția *fetch()* poate fi apelată și folosind un parametru de tip Request.
- **`Request()`** primește aceeași parametrii ca și funcția **`fetch()`**.

```
var myHeaders = new Headers();

var myInit = { method: 'GET',
               headers: myHeaders,
               mode: 'cors',
               cache: 'default' };

var myRequest = new Request('some.json', myInit);

fetch(myRequest).then(function(response) {
  return response.json();
}).then(function(myData) {
    console.log(myData);
 });
```

- Observație: Parametrii de tip query trebuie adăugați explicit la cerere (url, antet). Nu există metode speciale care permit transmiterea acestora.

Antete Fetch

- ***`Headers`*** permite crearea/păstrarea/accesarea antetelor unei cereri/unui răspuns.

- Este un dicționar de perechi (nume-antet, valoare-antet):

```
var content = "Hello World";
var myHeaders = new Headers();
myHeaders.append("Content-Type", "text/plain");
myHeaders.append("Content-Length", content.length.toString());
myHeaders.append("X-Custom-Header", "ProcessThisImmediately");
```

- Se poate transmite și un tablou conținând valorile antetelor:

```
myHeaders = new Headers({
  "Content-Type": "text/plain",
  "Content-Length": content.length.toString(),
  "X-Custom-Header": "ProcessThisImmediately",
});
console.log(myHeaders.has("Content-Type")); // true
console.log(myHeaders.has("Set-Cookie")); // false
myHeaders.set("Content-Type", "text/html");
myHeaders.append("X-Custom-Header", "AnotherValue");

console.log(myHeaders.get("Content-Length")); // 11
console.log(myHeaders.get("X-Custom-Header")); // ["ProcessThisImmediately", "AnotherValue"]

myHeaders.delete("X-Custom-Header");
console.log(myHeaders.get("X-Custom-Header")); // [ ]
```

Fetch - Body

- Cererile și răspunsurile pot conține informație/date.
- Informația poate fi de următoarele tipuri:
  - ArrayBuffer
  - ArrayBufferView
  - Blob/File
  - string
  - URLSearchParams
  - FormData
- Sunt definite metode pentru obținerea informației dintr-un răspuns. Toate metodele returnează un obiect de tip promise, care conțin informația.
  - arrayBuffer()
  - blob()
  - json()
  - text()
  - formData()

Fetch - Feature detection

- Suportul pentru Fetch API poate fi detectat prin verificarea existenței (în fereastră sau scopul workerului) a obiectelor:
  - Headers
  - Request
  - Response
  - funcția fetch().

Exemplu

```
if (self.fetch) {
    // execuția cererii fetch
} else {
    // folosirea XMLHttpRequest?
}
```

React JS

- React este o bibliotecă JavaScript declarativă, flexibilă și eficientă care permite construirea ușoară și rapidă a interfețelor cu utilizatorul.
- Permite dezvoltatorilor să creeze aplicații web mari, care folosesc date ce se pot modifica în timp, fără a reîncărca pagina.
- Obiectivele principale: rapiditate, simplitate, scalabilitate.
- React procesează doar interfețe grafice în aplicații.
    - Corespunde View-ului din șablonul Model-View-Controller (MVC) și poate fi folosit împreună cu alte biblioteci sau frameworkuri JavaScript din MVC (ex. AngularJS).
- Este dezvoltat și întreținut de o comunitate formată din dezvoltatori de la Facebook, Instagram și dezvoltatori individuali.
- În prezent este folosit pentru site-uri precum Netflix, Airbnb, Walmart, etc.

React - Elemente

- Elementele React sunt cele mai mici construcții ale unei aplicații React. Un element descrie ceea ce trebuie afișat pe ecran.
- Elementele React sunt obiecte simple și ușor de creat. React DOM se ocupă de actualizarea DOM pentru potrivirea cu elementele React.

```
const element = <h1>Hello, world</h1>;
```

- Redarea unui element în DOM: `<div id="root"></div>`
- Este numit un nod DOM, pentru că tot ce este conținut de el va fi gestionat de React DOM.
- Aplicațiile dezvoltate folosind React conțin de obicei un singur nod root.
- Pentru redarea unui element React într-un nod root, se apelează: *ReactDOM.render()*:

```
const element = <h1>Hello, world</h1>;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

- Actualizarea unui element: elementele React sunt *immutable.* Un element odată creat, datele sale nu se mai pot modifica.
- Majoritatea aplicațiilor React apelează ReactDOM.render() o singură dată.
- Biblioteca React actualizează la afișare doar datele modificate. React DOM compară un element și descendenții săi cu datele anterioare și invocă doar actualizarile necesare pentru ca DOM să aibă starea dorită.

JSX

```
//nu este nici un string, nici HTML.
const element = <h1>Hello, world!</h1>;
```

- Sintaxa este numită **JSX**, și este o extensie a sintaxei JavaScript.
- Este modul recomandat de descriere în React a interfețelor grafice.
- JSX produce elemente React.
- Se pot include orice expresii JavaScript în JSX, prin includerea lor între acolade(‘{‘ ‘}’)

```
const user = {
  firstName: 'Popescu',
  lastName: 'Ana'
};

function formatName(user) {
  return user.firstName + ' ' + user.lastName;
}

const element = (
  <h1>
    Hello, {formatName(user)}!
  </h1>
);

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

Specificarea atributelor folosind JSX

- Se pot folosi ghilimelele pentru specificarea atributelor de tip string:

```
const element = <div tabIndex="0"></div>;
```

- Se folosesc acoladele pentru includerea unei expresii JavaScript ca și valoarea a unui atribut:

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

- Nu se pun ghilimele când se includ expresii JavaScript (expresiile vor fi considerate stringuri)

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

- Dacă un tag este gol, se închide cu />, asemănător XML:

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

- Tagurile JSX pot conține fii:

```
const element = (
  <div>
    <h1>Salut</h1>
    <h2>Bine ai venit!</h2>
  </div>
);
```

Componente React

- React are la bază definirea unor componente React.
- O componentă poate păstra mai multe instanțe a altei/altor componente (relația părinte-copil).
- Componentele permit împărțirea UI în parți independente și reutilizabile, și dezvoltarea independentă a acestora.
- Definirea unei componente: folosind clasa `React.Component` (clasa JavaScript, ES6).

```
class Greeting extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

- `React.Component` este o clasă abstractă.
- Componentele definite de dezvoltator redefinesc cel puțin metoda *render()*, care specifică modul de afișare a componentei.
- Numele componentelor încep întotdeauna cu litera mare.
- Exemplu, `<div />` reprezintă un tag DOM, dar `<Greeting />` reprezintă o componentă și este necesar ca *Greeting* să fie disponibil în scop.
- Datele afișate de o componentă React sunt obținute prin proprietăți (*this.props*) sau din starea componentei (*this.state*).

Ciclul de viață

- Fiecare componentă are câteva metode ce sunt apelate în ciclul său de viață.
- Aceste metode pot fi redefinite pentru a particulariza componenta.
- Metodele care încep cu '*will*' sunt apelate înaintea apariției evenimentului, iar cele cu '*did*' sunt apelate după apariția evenimentului.
- *Crearea/asamblare/afișare*: metodele sunt apelate când o componentă este creată și adăugată la DOM.
    - *constructor(), componentWillMount(), render(), componentDidMount()*
- *Actualizare*: poate fi cauzată de modificări ale proprietăților sau a stării componentei. Metodele sunt apelate când componenta este re-afișată.
    - *componentWillReceiveProps(), shouldComponentUpdate(), componentWillUpdate(), render(), componentDidUpdate()*
- *Dezasamblarea*: metoda este apelată când componentă este ștearsă din DOM:
    - *componentWillUnmount()*

Componente React

- Alte metode
  - *setState()*
  - *forceUpdate()*
- Proprietăți ale clasei
  - *defaultProps*
  - *displayName*

Componente și Props

- Componentele React au două tipuri de date: starea și proprietățile.
- Când React întâlnește un element reprezentând o componentă definită de dezvoltator, îi transmite atributele JSX ca și un singur obiect numit “props”.
- *this.props* - conține proprietățile definite de componenta care a apelat această componentă.
- *this.props.children* este o proprietate specială, conținând tagurile fii ale componentei curente.
- Proprietățile sunt *read-only*, o componentă nu trebuie să-și modifice propriile proprietăți:

```
ReactDOM.render(
  <Greeting name=“Ana”>,
  document.getElementById('root')
);
```

```
<button onClick={this.handleClick}>
  Say hello
</button>
```

Compunerea componentelor

- Componentele pot referi alte componente în funcția render().
- Permite folosirea aceluiași nivel de abstractizare a componentelor: un buton, un form, un dialog, etc sunt toate exprimate ca și componente.
- Exemplu, putem crea o componentă care afișează Greeting de mai multe ori:

```
class App extends React.Component {
  render() {
    return (
    <div>
      <Greeting name="Ana" />
      <Greeting name="Maria" />
      <Greeting name="Ion" />
    </div> );
 }
}

 ReactDOM.render(
  <App/>,
  document.getElementById('root')
);
```

React Component -States

- Starea (eng. state) unei componente este informația pe care componenta se așteaptă să o gestioneze singură.
- *this.state* - conține date specifice componentei, care se pot modifica în timp. Starea este definită de dezvoltator, și ar trebui să fie un obiect JavaScript normal.
- Dacă informația nu este folosită în funcția *render()*, atunci nu face parte din starea componentei.
- Obținerea stării se face folosind *this.state*.
- *This.state* trebuie considerată immutable, și nu trebuie modificată în mod direct.
- Orice actualizare a stării componentei se face folosind funcția *this.setState()*. De fiecare dată când *this.setState()* este apelat, React actualizează starea, determină diferențele dintre starea anterioară și noua stare și injectează o mulțime de modificări DOM-ului corespunzător paginii. În acest fel actualizările UI sunt rapide și eficiente.

React Component -States

- Recomandarea este ca inițializarea stării să se facă în constructor cu valori implicite, iar apoi în metoda *componentDidMount()* să se actualizeze starea cu datele obținute de la server (modelul aplicației).
- Din acel moment, actualizările vor fi determinate de acțiunile utilizatorului sau de alte evenimente.

```
class Clock extends React.Component {
  constructor(props) {
    super(props);   //apelul constructorului clasei de baza folosind props
    this.state = {date: new Date()};
  }

render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
  componentDidMount() {
   //va fi apelată după adăugarea componentei la DOM
  }

  componentWillUnmount() {
  //eliberarea resurselor folosite, dacă există
  }
}
```

React Component -States

- Observații legate de setState():
- NU se modifică starea direct.
  - Exemplu: nu va provoca re-afișarea unei componente:

```
this.state.comment = 'Hello';
```

  - Ar trebui să se folosească metoda *setState()*

```
// Correct
this.setState({comment: 'Hello'});
```

  - Doar în constructor se poate seta informația folosind direct *this.state.*
- Actualizările stării pot fi asincrone.
  - React poate decide ca mai multe apeluri ale funcției *setState* să fie executate împreună, pentru a mări performanța.
  - Deoarece *this.props* și *this.state* pot fi actualizate asincron, nu ar trebui să ne bazăm pe valorile lor când se calculează noua valoare.

```
// Greșit
this.setState({
  counter: this.state.counter + this.props.increment,
});
```

  - Soluția corectă: folosirea metodei setState care primește 2 parametri: starea anterioară și valorile proprietăților în momentul actualizării:

```
// Corect
this.setState((prevState, props) => ({
  counter: prevState.counter + props.increment
}));
```

React Component -States

- Actualizările stării sunt unificate:
  - La apelul metodei *setState()*, React reunește obiectul primit ca și parametru cu starea curentă. Ex. starea conține mai multe variabile independente:

```
constructor(props) {
  super(props);
  this.state = {
    posts: [],
    comments: []
  };
}
```

  - Variabilele se pot actualiza independent, folosind apeluri diferite ale funcției setState():

```
componentDidMount() {
  fetchPosts().then(response => {
    this.setState({
      posts: response.posts
    });
  });

  fetchComments().then(response => {
    this.setState({
      comments: response.comments
    });
  });
}
```

  - Reuniunea este superficială: *this.setState({comments})* va lăsa *this.state.posts* nemodificat, dar va înlocui *this.state.comments*.

React Component -States

- Datele sunt transmise de la componenta părinte la descendenți.
  - Nici părinții, nici descendenții nu știu dacă o anumită componentă este *stateful* sau *stateless*.
  - Adesea, starea este numită stare locală sau încapsulată. Starea nu este disponibilă altor componente, doar componentei care a creat-o.
  - O componentă poate alege să transmită starea sa componentelor fii prin intermediul proprietăților:

```
<h2>It is {this.state.date.toLocaleTimeString()}.</h2>
<FormattedDate date={this.state.date} />
```

    - Acest mod de a transmite informația este numit flux de date "top-down" sau "unidirectional". Orice stare este totdeauna definită de o anumită componentă, și orice informație sau UI derivat din acea stare poate afecta doar componentele descendente din arbore.
- În aplicațiile React, faptul că o componentă este *stateful* sau *stateless* este considerat detaliu de implementare care se poate schimba în timp. Componente stateless pot conține componente stateful și invers.

Tratarea evenimentelor

- Asemănătoare cu tratarea evenimentelor corespunzătoare elementelor DOM.
- Diferențe:
  - Evenimentele React sunt numite folosind camelCase, nu doar litere mici.
  - Cu JSX se poate transmite o funcție ca și un event handler.

    HTML:

```
<button onclick="activateLasers()">
  Activate Lasers
</button>
```

    React:

```
<button onClick={activateLasers}>
  Activate Lasers
</button>
```

  - Nu se poate returna *false* pentru preveni comportamentul implicit din React. Trebuie apelată explicit metoda *preventDefault*.

HTML: pentru a preveni comportamentul implicit de a deschide o nouă pagină:

```
<a href="#" onclick="console.log('The link was clicked.'); return false">
  Click me
</a>
```

React:

```
function ActionLink() {
  function handleClick(e) {
    e.preventDefault();
    console.log('The link was clicked.');
  }

  return (
    <a href="#" onClick={handleClick}>
      Click me
    </a>
  );
}
```

Tratarea evenimentelor

- La definirea unei componente folosind o clasă ES6, adesea handlerul pentru evenimente este definit ca și o metodă a clasei.

```
class Toggle extends React.Component {
  constructor(props) {
    super(props);
    this.state = {isToggleOn: true};
    // Asociere necesară pentru a putea folosi this in callback
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    this.setState(prevState => ({
      isToggleOn: !prevState.isToggleOn
    }));
  }
  render() {
    return (
      <button onClick={this.handleClick}>
        {this.state.isToggleOn ? 'ON' : 'OFF'}
      </button>
    );
  }
}
ReactDOM.render(
  <Toggle />,
  document.getElementById('root')
);
```

- Semnificația cuvântului *this* în JSX callbacks. În JavaScript, metodele unei clase nu sunt asociate implicit. Dacă asocierea (*binding*) nu se face explicit, obiectul *this* va fi considerat ca undefined când funcția va fi apelată.
- Comportament specific limbajului JavaScript.
- În general, dacă ne referim la o metodă fără a folosi () după ea, cum ar fi onClick={this.handleClick}, metoda trebuie asociată.

Tratarea evenimentelor

- O altă modalitate de asociere este folosirea sintaxei de initializare a proprietăților:

```
class LoggingButton extends React.Component {
  // Asigură asocierea.
  // Sintaxa  *experimentala* .
  handleClick = () => {
    console.log('this is:', this);
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        Click me
      </button>
    );
  }
}
```

Input Forms

```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: ''};

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  handleSubmit(event) {
    alert('A name was submitted: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <input type="text" value={this.state.value} onChange={this.handleChange} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

Create-react -App

- Instalare: *NodeJs:* https://nodejs.org/en/
    - npm: package manager (inclus in nodejs)
- Instalare: *create-react-app*
        - npm install -g create-react-app
        - create-react-app my-app (Numele directorului cu litere mici!)
        - cd my-app
        - npm start (localhost:3000)
- Nu e necesară configurarea folosind babel, webpack/browserify, typescript, etc.

Referințe

- Jake Archibald, *JavaScript Promises: an Introduction*,

https://developers.google.com/web/fundamentals/getting-started/primers/promises

- Using Fetch:

https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

- Matt Gaunt, *Introduction to fetch()*,

https://developers.google.com/web/updates/2015/03/introduction-to-fetch

- Documentație React, https://facebook.github.io/react/