Medii de proiectare și programare

2021-2022
Curs 10

Conținut

- REST
  - Clienti Java/C#
- Servicii REST - Clienti web
  - JavaScript:
    - Promise
    - Fetch

Clienți REST

- API pentru apelarea unui serviciu REST:
  - Java
  - .NET
  - JavaScript
  - etc.
- Browser web
- Extensii ale browserelor web:
  - RESTED
  - POSTMAN
  - Advanced REST Client
  - etc.

Spring REST Client

- RestTemplate definește 36 de metode pentru interacțiunea cu resurse REST.

Method	Description
`delete()`	Performs an HTTP `DELETE` request on a resource at a specified URL
`exchange()`	Executes a specified HTTP method against a URL, returning a `ResponseEntity` containing an object mapped from the response body
`execute()`	Executes a specified HTTP method against a URL, returning an object mapped from the response body
`getForEntity()`	Sends an HTTP `GET` request, returning a `ResponseEntity` containing an object mapped from the response body
`getForObject()`	Sends an HTTP `GET` request, returning an object mapped from a response body
`headForHeaders()`	Sends an HTTP `HEAD` request, returning the HTTP headers for the specified resource URL
`optionsForAllow()`	Sends an HTTP `OPTIONS` request, returning the `Allow` header for the specified URL
`postForEntity()`	`POST`s data to a URL, returning a `ResponseEntity` containing an object mapped from the response body
`postForLocation()`	`POST`s data to a URL, returning the URL of the newly created resource
`postForObject()`	`POST`s data to a URL, returning an object mapped from the response body
`put()`	`PUT`s resource data to the specified URL

Spring REST Client

```
public class StartRestClient {

    public static void main(String[] args) {
        RestTemplate restTemplate=new RestTemplate();

     //Adding an article
        Article article=new Article(“AB CB","AI","Genetic");
        try{
        String articleId= restTemplate.postForObject("http://localhost:8080/
conference/articles",article, String.class);

        // Updating  article ...
        article.setTitle("New title");
        restTemplate.put("http://localhost:8080/conference/articles/"+articleId,
article);

       }catch(RestClientException ex){
            System.out.println("Exception ... "+ex.getMessage());
        }

    }
}
```

.NET REST Client

- Clasa **`System.Net.Http.HttpClient`**
  - Trebuie adăugat (folosind NuGet) pachetul **`Microsoft.AspNet.WebApi.Client`**
  - HttpClient ar trebui instanțiat o singură dată într-o aplicație și refolosit pe parcursul rulării aplicației. Crearea unei noi instanțe pentru fiecare cerere, poate genera erori de tip SocketException (se atinge numărul maxim de conexiuni permise).

```
class MainClass{
  static HttpClient client = new HttpClient();
  public static void Main(string[] args){
     client.DefaultRequestHeaders.Accept.Clear();
     client.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderValue("application/json"));
      // Get an article …;
      Article result = await GetArticleAsync(“http://localhost:8080/conference/articles/art124");
}
   static async Task<Article> GetArticleAsync(string path){
              Article article = null;
              HttpResponseMessage response = await client.GetAsync(path);
              if (response.IsSuccessStatusCode)
              {
                   article = await response.Content.ReadAsAsync<Article>();
              }
              return article;
  }
}
```

Referințe

- https://docs.microsoft.com/en-us/aspnet/web-api/overview/advanced/calling-a-web-api-from-a-net-client
- Building REST services with Spring:

  https://spring.io/guides/tutorials/bookmarks/

  https://spring.io/guides/tutorials/rest/
- http://www.service-architecture.com/articles/web-services/
- Craig Walls, Spring in Action, 4th Edition, Ed. Manning, 2014
- Alte tutoriale

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-ul 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 execuție, 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 callback 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.

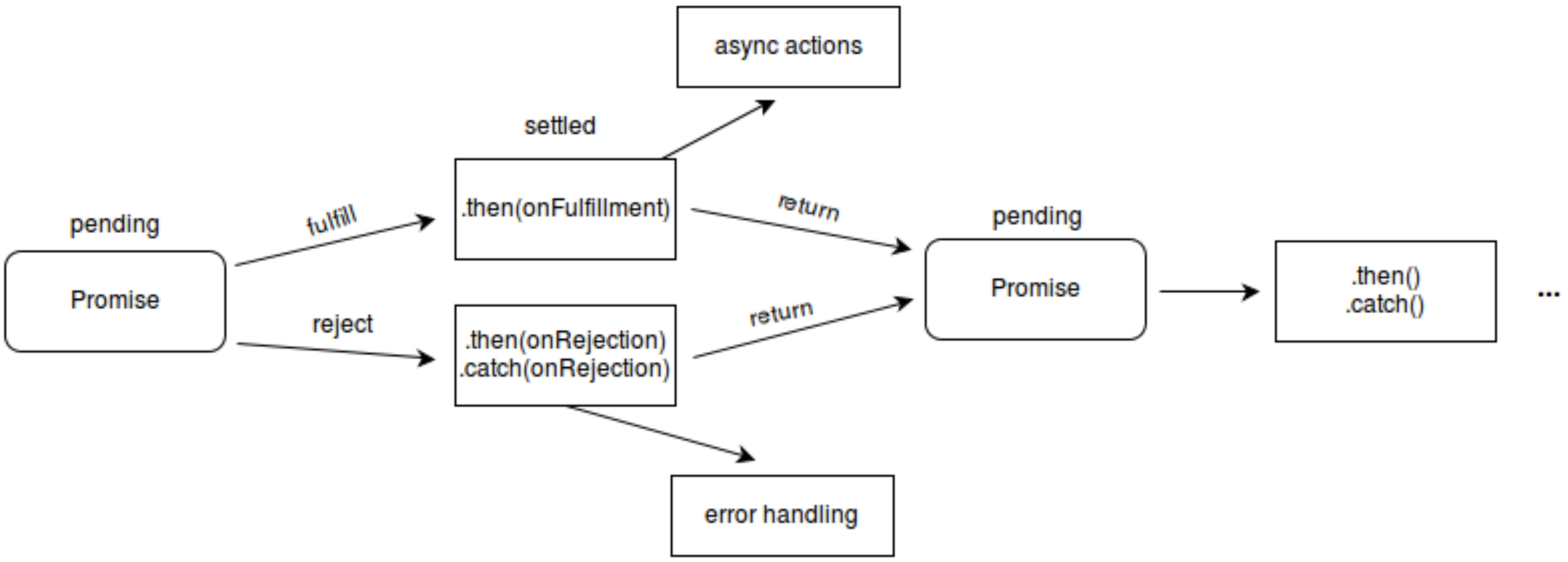
```
var promise = new Promise(function(resolve, reject) {
  resolve(1);
});

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

// ———

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

Înlănțuire Promises (*chaining*)



JavaScript - Promises chaining

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

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

```
get('url/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.

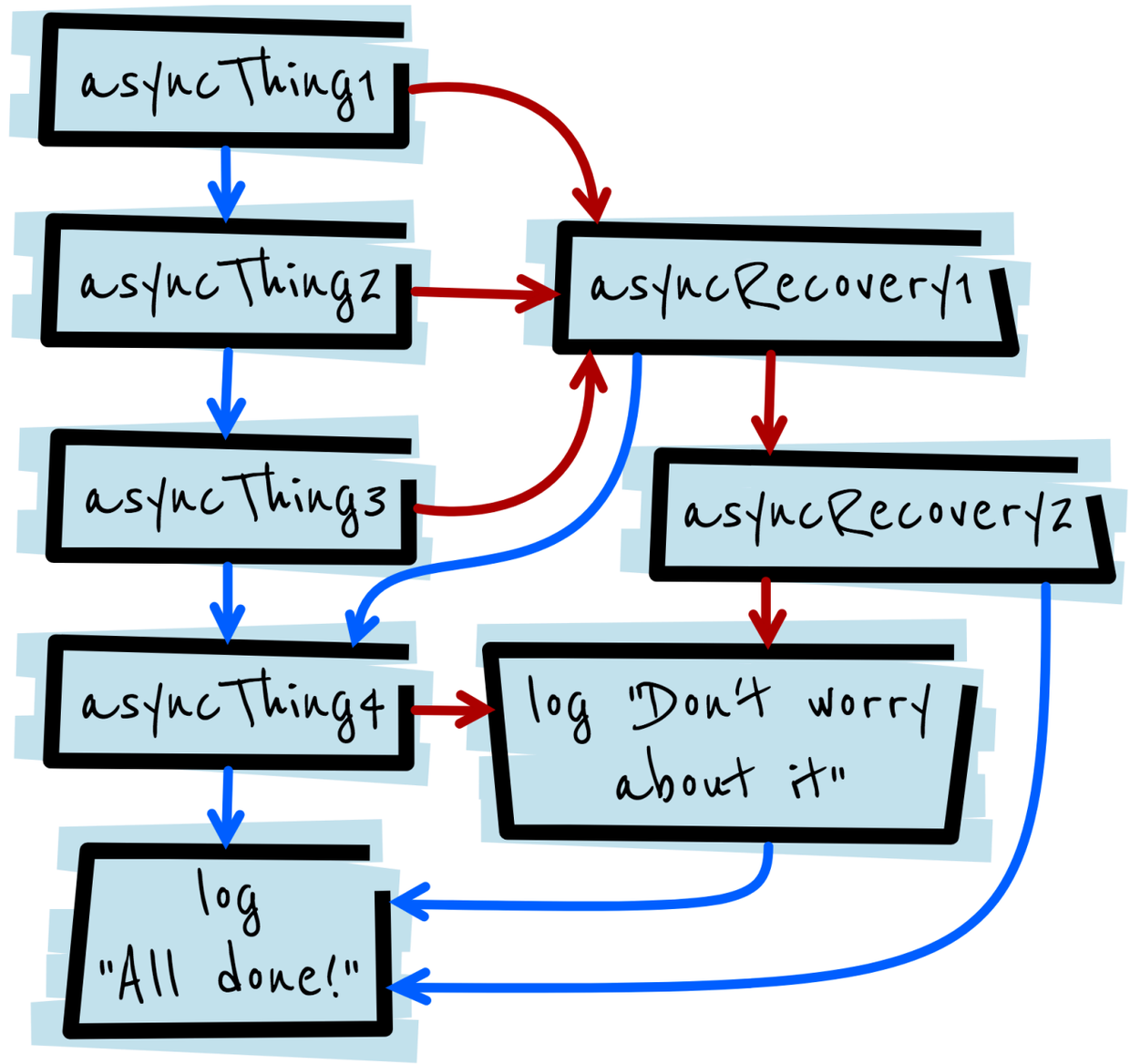
- Cele două exemple de cod, nu sunt echivalente (ultimul este echivalent cu):

```
get('url/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 Promises async/await

- *async* definește o funcție asincronă. Rezultatul funcției va fi implicit un Promise.

```
async function name([param[, param[, ... param]]]) {
   statements
}
```

- Operatorul *await* așteaptă rezultatul execuției unui Promise. Poate fi folosit doar în interiorul unei funcții asincrone.

```
[value] = await expression;
```

- *expression* poate fi un Promise sau orice valoare
- Dacă *expression* nu este un Promise, ea este convertită folosind Promise.resolved(*expression*)
- *value* va primi rezultatul execuției cu succes a Promise-ului sau valoarea expresiei (dacă expresia nu este un Promise)
- Dacă Promise-ul este respins, expresia *await* aruncă excepție cu valoarea respingerii.

JavaScript Promises async/await

```
function resolveAfter2Seconds(x) {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve(x);
    }, 2000);
  });
}

async function f1() {
  var x = await resolveAfter2Seconds(10);
  console.log(x); // 10
}

f1();
```

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.

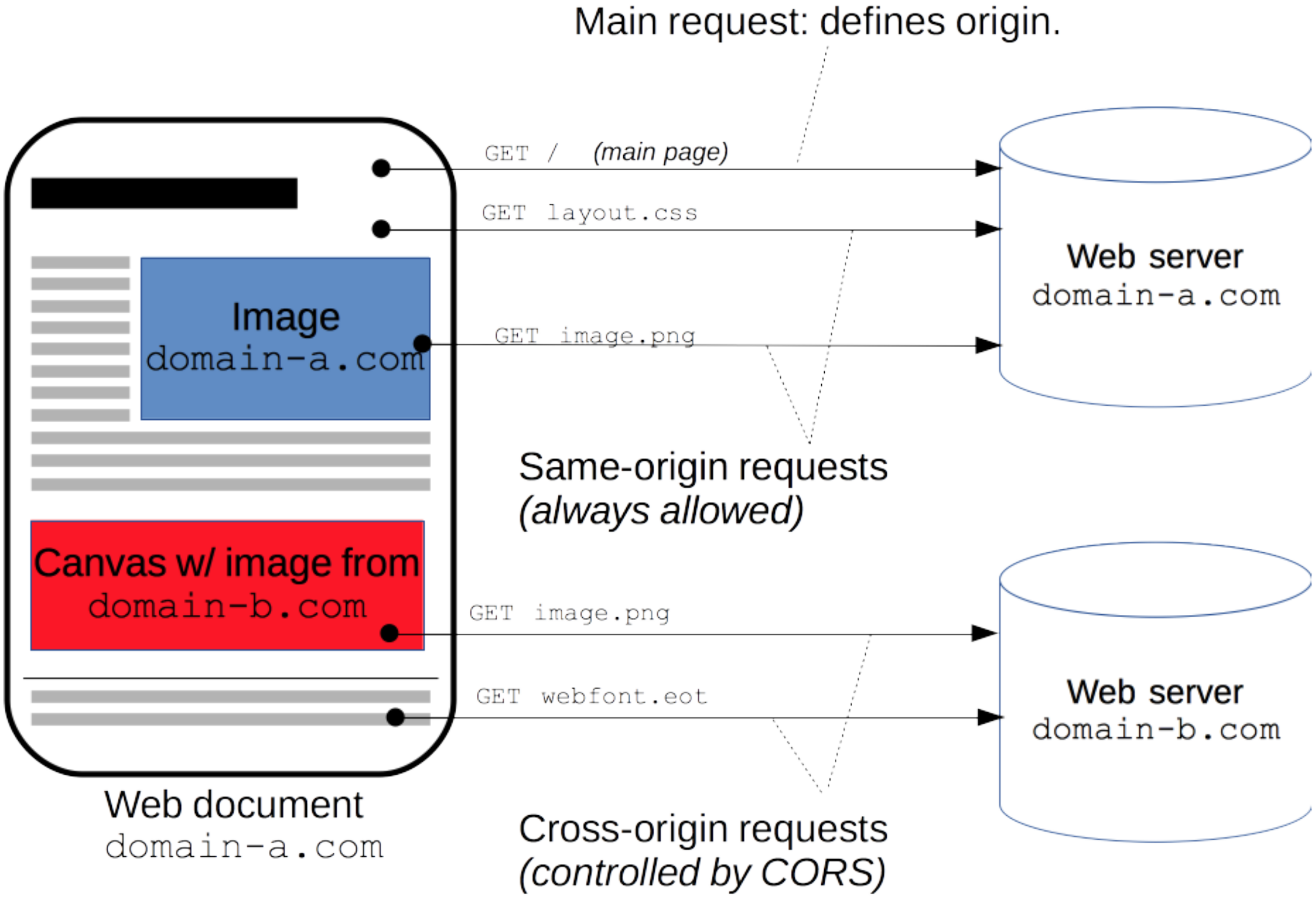
Fetch Response

- Apelul funcției *fetch* returnează un obiect 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 respectiv (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