

SUPSI

Introduzione

Patrick Ceppi

Web applications 2

Dove ci siamo lasciati

- **Applicazione web di ticketing**
 - risposte HTTP **REST** CRUD
 - risposte che ritornano direttamente HTML (thymeleaf, **Server Side Rendering SSR**)
 - **ricerca di ticket via javascript:**
 - richiesta asincrona ritorna un JSON con i ticket trovati (ajax)
 - DOM viene manipolato
 - in pratica bisogna ricreare dei pezzi HTML uguali a quelli creati con thymeleaf (*template engine* lato client, **Client Side Rendering CSR**)

Dove ci siamo lasciati

- Applicazione web di ticketing
- Thymeleaf, esempio *index-card.html*

```
<article class="col-sm-6 col-md-4" th:each="ticket : ${tickets}">
  <div class="card mb-4 shadow-sm">
    <div class="card-body">
      <p style="color:grey">
        <span th:text="'#'+${ticket.id}"></span>
        <span class="badge bg-primary detail-status" th:text="${ticket.status}">open</span>
        <strong th:text="${ticket.type.name}">bug</strong> | <span th:text="${#dates.format(ticket.date, 'HH:m"

      </p>
      <strong><span class="card-title" th:text="${ticket.title}">Ticket title</span></strong>
      <hr>
      <p class="card-description"><span th:text="${ticket.description}">Ticket description</span></p>
      <div class="d-flex justify-content-between align-items-center">
        <div class="btn-group">
          <a class="btn btn-sm btn-outline-secondary" th:href="@{'/ticket/'+${ticket.id}}">View</a>
          <a class="btn btn-sm btn-outline-secondary" sec:authorize-url="/ticket/*/edit" th:href="@{'/ticke"

        </div>
      </div>
    </div>
  </div>
</div>
</article>
```

Dove ci siamo lasciati

- Applicazione web di ticketing
- Javascript, esempio *ajaxSearch.js*

```
articles +=
'<article class="col-sm-6 col-md-4">\n'+
'  <div class="card mb-4 shadow-sm">\n'+
'    <div class="card-body">\n'+
'      <p style="color:grey">\n'+
'        <span>#'+tickets[i].id+'</span>\n'+
'        <span class="badge bg-primary detail-status">'+tickets[i].status+'</span>\n'+
'        <strong>'+tickets[i].type+'</strong> | <span>'+dateFormatted+'</span> by <a href="#">'+ticke
'      </p>\n'+
'      <strong><span class="card-title">'+tickets[i].title+'</span></strong>\n'+
'      <hr>\n'+
'      <p class="card-description"><span>'+tickets[i].description+'</span>\n'+
'      </p>\n'+
'      <div class="d-flex justify-content-between align-items-center">\n'+
'        <div class="btn-group">\n'+
'          <a class="btn btn-sm btn-outline-secondary" href="'+context+'ticket/'+tickets[i].id+
'        </div>\n'+
'      </div>\n'+
'    </div>\n'+
'  </div>\n'+
'</article>\n';
```

Dove ci siamo lasciati

- **Applicazione web di ticketing**

- ✓ La ricerca è **interattiva**, molto **veloce**
- ✓ Il risultato della ricerca dei ticket (JSON) è **disaccoppiato** da come i risultati sono visualizzati nella pagina (*template* lato client)
- ✗ **Bookmarks** non funzionano
- ✗ Bottoni del browser **avanti/indietro** non funzionano correttamente
- ✗ La pagina **non è indicizzabile** dai web crawler come google
- ✗ Abbiamo lo stesso **template scritto due volte**, con due tecnologie diverse

Non sono due fattori importanti per gli utenti, ma per gli sviluppatori

Dove ci siamo lasciati

- **Applicazione web di ticketing**
 - **L'optimum** sarebbe **dare all'utente**:
 - delle **pagine fluide**, ma senza perdere le funzionalità base del browser (**bookmarks, avanti/indietro**) e
 - la possibilità di poter essere **indicizzati** dai web crawlers
 - mentre nella **parte di sviluppo** dell'applicazione:
 - avere un'architettura il più **disaccoppiata** possibile, e
 - **non** avere del **codice duplicato** (sia di logica applicativa che di logica di interfaccia grafica e di interazione con l'utente)

Ajax

"Desktop applications have a richness and responsiveness that has seemed out of reach on the Web. The same simplicity that enabled the Web's rapid proliferation also creates a gap between the experiences we can provide and the experiences users can get from a desktop application."

Jesse James Garrett nel **2005**, quando definì il termine **Ajax**

Nota: XMLHttpRequest fu incluso in Internet Explorer nel **1999**

Single Page Application (SPA)

- Javascript (DOM, Ajax) è uno strumento fondamentale per creare delle pagine web, non solo **interattive**, ma che sempre più **assomigliano alle applicazioni desktop** per quanto riguarda la velocità di risposta (di refresh) e la percezione dell'utente quando le utilizza
- Nel **2004** Google Maps, Gmail, e altre, hanno dimostrato che si potevano creare delle web apps con un ottimo design dell'interfaccia grafica utilizzando ajax, senza mai ricaricare la pagina e creando un'esperienza per l'utente totalmente diversa rispetto a web apps precedenti
- Sono stati i primi esempi di **Single Page Application SPA**

Single Page Application (SPA)

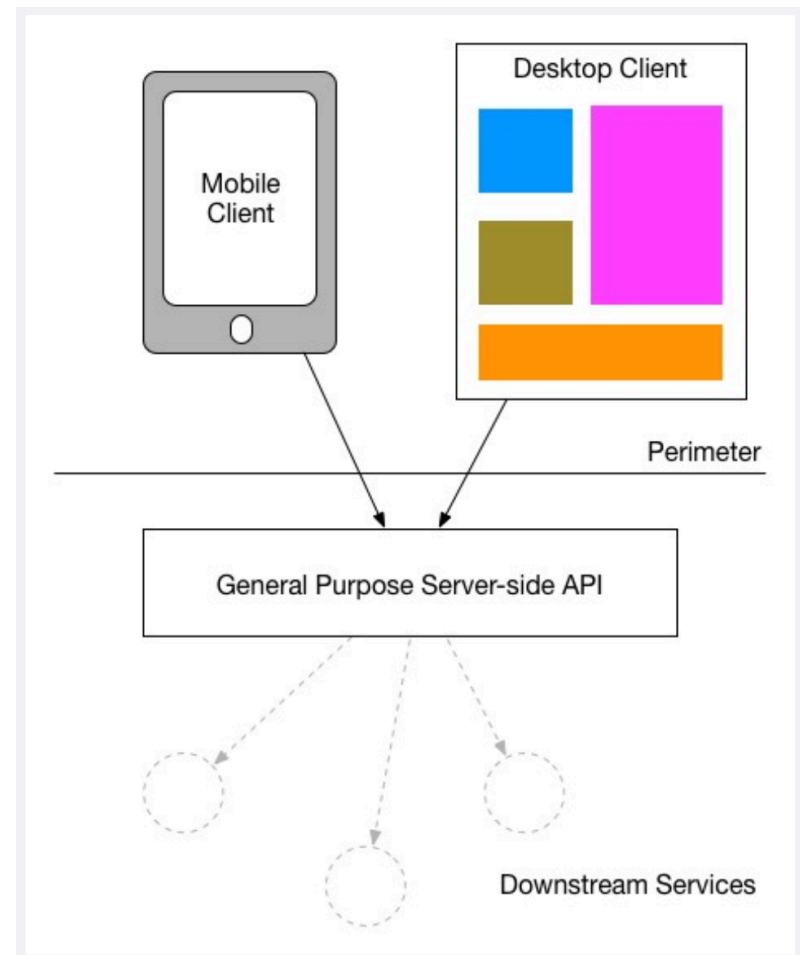
- È un'applicazione web o un sito web che può essere usato o consultato su una singola pagina web con l'obiettivo di fornire una **esperienza utente più fluida e simile alle applicazioni desktop** dei sistemi operativi tradizionali.
- In una SPA
 - tutto il codice necessario (HTML, JavaScript e CSS) è recuperato in un **singolo caricamento** della pagina
 - le **risorse** appropriate sono **caricate dinamicamente**(ajax) e aggiunte alla pagina quando necessario,
 - di solito **in risposta ad azioni dell'utente**

Single Page Application (SPA)

- In una SPA la creazione della pagina, del HTML, avviene lato client
 - ci vogliono quindi delle tecnologie di **templating** lato client
 - ma non solo, bisogna gestire il **routing**
 - e **gestire lo stato dell'applicazione** lato client
- E persistono le problematiche: bookmarks, avanti/indietro, indicizzazione, ...
- Sono molti i frameworks e librerie che ci permettono di sviluppare applicazioni SPA: Angular, Knockout.js, React, Vue.js, Ember.js, ...

Disaccoppiamento

- Per **disaccoppiare la parte logica dall'interfaccia grafica**, il backend non deve contenere nessuna logica legata alla GUI, i.e. non deve ritornare risposte HTTP con HTML
- La parte **backend espone un API REST** che i vari client possono utilizzare per mostrare i dati nel modo più idoneo
- La **parte client** deve avere un meccanismo di **templating**
- Addirittura si potrebbe pensare di avere più backend, ognuno specifico per un determinato tipo di client



Disaccoppiamento

- Il disaccoppiamento porta diversi **benefici**:
 - maggiore **flessibilità nello sviluppo** della User Interface
 - **velocità nel rilascio di funzionalità** frontend, senza restrizioni imposte dal backend
 - **scalabilità** del backend indipendente dalla User Interface
 - **separazione** delle codebases, **minore complessità** dell'architettura e della gestione dei teams
- Se vogliamo avere un'architettura disaccoppiata nello sviluppo di applicazioni web, ci troveremo sicuramente a dover sviluppare del codice javascript per gestire la parte di rendering del HTML

Javascript templating

- Ci sono librerie javascript, come **mustache** (mustache.github.io), che ci permettono di avere un *template engine* nel browser
- Esempio:

Template

```
{{#items}}  
  <li><a href="{{url}}">{{name}}</a></li>  
{{/items}}
```

JSON

```
{  
  "items": [  
    {"name": "red", "url": "#Red"},  
    {"name": "green", "url": "#Green"}  
  ]  
}
```

Risultato

```
<li><a href="#Red">red</a></li>  
<li><a href="#Green">green</a></li>
```

- Mustache è della fine del **2009**

KnockoutJS

- Libreria del **2010** che implementa il pattern Model-View-ViewModel (MVVM)
- Il pattern MVVM prevede una netta separazione tra dati (Model), interfaccia utente (View) e la modalità con la quale i dati vengono rappresentati (ViewModel)
- Esempio:

```
<p>First name: <input data-bind="value: firstName" /></p>  
<p>Last name: <input data-bind="value: lastName" /></p>  
<p>Full name: <strong data-bind="text: fullName"></strong></p>
```

```
function AppViewModel() {  
    this.firstName = ko.observable("Marco");  
    this.lastName = ko.observable("Bernasconi");  
  
    this.fullName = ko.computed(function() {  
        return this.firstName() + " " + this.lastName();  
    }, this);  
}
```

```
ko.applyBindings(new AppViewModel());
```

Se l'utente modifica il
valore dei campi, il
valore di FullName
cambia automaticamente

First name:

Last name:

Full name: **Marco Bernasconi**

React

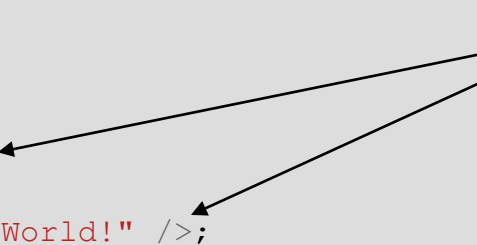
- Già nel 2011, gli sviluppatori di Facebook hanno iniziato ad affrontare alcuni problemi di manutenzione del codice
- Più le funzionalità delle loro applicazioni crescevano, più i loro team crescevano
- Facebook Ads è stata la prima applicazione ad avere problemi: era diventata **difficile da gestire** e gli **sviluppi rallentavano**
- Attorno al 2012 Jordan Walke, un ingegnere di Facebook, creò **React** proprio per risolvere questi problemi
- React può essere uno strumento molto efficace quando si hanno GUI complesse
 - <https://www.youtube.com/watch?v=KVZ-P-ZI6W4&t=335>
- Prima release ufficiale del **2013**

Esempio React

HTML, JSX (JavaScript XML) e JavaScript

```
<div id="myReactApp"></div>

<script type="text/babel">
  function Greeter(props) {
    return <h1>{props.greeting}</h1>
  }
  var App = <Greeter greeting="Hello World!" />;
  ReactDOM.render(App, document.getElementById('myReactApp'));
</script>
```

A diagram with a box labeled "JSX" in the top right. Two arrows originate from this box: one points to the JSX element `<h1>{props.greeting}</h1>` inside the `Greeter` function, and the other points to the JSX element `<Greeter greeting="Hello World!" />` in the `App` variable assignment.

Risultato

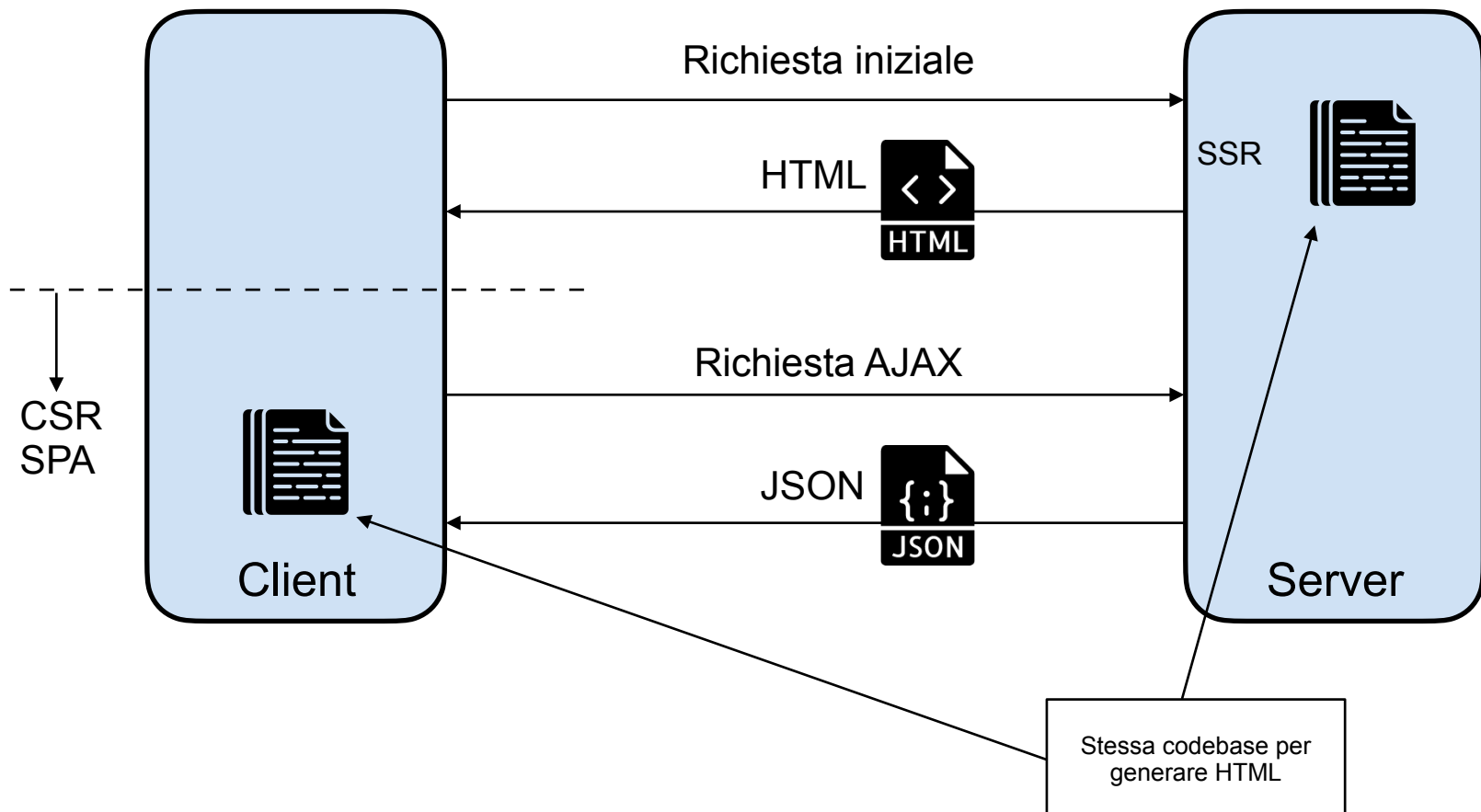
```
<div id="myReactApp">
  <h1>Hello World!</h1>
</div>
```

- React si occupa solo del rendering dei dati sul DOM, e quindi la creazione di applicazioni React richiede generalmente l'uso di librerie aggiuntive per la gestione dello stato e del routing.

Isomorphic/Universal application

- Abbiamo visto come esistono diversi framework per sviluppare applicazioni web in cui la GUI è completamente renderizzata lato client, ma abbiamo ancora alcuni problemi, uno su tutti l'indicizzazione da parte dei web crawlers
- **È possibile utilizzare la stessa tecnologia lato client anche lato server**, e generare la stessa interfaccia grafica?
- Lato client possiamo usare esclusivamente javascript, quindi abbiamo bisogno di usare javascript anche lato server per ottenere una applicazione che possa utilizzare la stessa tecnologia in entrambe le parti.
- **Node.js** è di gran lunga la scelta più comune per creare questo tipo di applicazioni, in quanto utilizza javascript lato server. Queste applicazioni sono chiamate **isomorfe** o universali.
- Non è comunque l'unica possibilità. Vedi: <https://www.baeldung.com/react-nashorn-isomorphic-app>

Isomorphic/Universal application



Conclusioni

- La maggior parte dei **framework**/librerie per lo sviluppo di **SPA** hanno risolto nel **frattempo le problematiche** riguardo l'indicizzazione, i bookmarks, avanti/indietro, ...
- Le applicazioni web hanno spesso un **architettura molto disaccoppiata**, dove lo stesso backend può servire sia applicazioni web che applicazioni native mobile o applicazioni desktop
- **Non esiste la soluzione che va bene per tutto**, anche il solo SSR è corretto. La complessità dell'interfaccia, l'interattività che devono avere le pagine, sono le maggiori caratteristiche da valutare per scegliere i framework e librerie idonee
- Nello sviluppo web è **fondamentale** comunque **sapere javascript** e quindi capirne alcune caratteristiche, che non si trovano negli altri linguaggi Object Oriented
- Caratteristiche **javascript da approfondire**: gestione eventi, promises, callback, programmazione funzionale, espressioni lambda, ...