TWEB 2023-2024

Indice

Jakarta Servlet

URL

Si distinguono due tipi di URL: statica e dinamica. La loro struttura è simile, con piccole differenze.

La URL statica identifica una risorsa pre-esistente nel web server, ed è così composta:

`protocol://domain[:port][/path]/resource[#fragment]`

- `protocol` : il protocollo utilizzato, tipicamente http o https;
- `//domain` : identifica il web server;
- `[:port]` : la porta di rete su cui il web server è in ascolto (il browser la stabilisce in automatico e non la mostra nella barra degli indirizzi, 80 per http e 443 per https);
- `[/path]` : la posizione della risorsa richiesta nel file system associato al web server;
- `/resource` : la risorsa richiesta, tipicamente un file (pdf, png, …);
- `[#fragment]` : opzionale, può indicare uno specifico punto nella risorsa richiesta.

La URL dinamica, invece, identifica una richiesta di elaborazione, ed ha la seguente struttura:

`protocol://domain[:port][/path]/resource[?querystring]`

Ciò che cambia rispetto alla URL statica è:

- `[/path]` + `/resource` : ci si riferisce a questa combinazione come ***route***, e identifica la web application che si occuperà dell'elaborazione e l'elaborazione richiesta;
- `[?querystring]` : opzionale, specifica i parametri dell'elaborazione come coppie attributo=valore. È possibile specificare più parametri in questo modo:

```
[resto dell'URL]?attr=val&attr=val&…
```

Pagine

Allo stesso modo le pagine web possono essere **statiche** (con URL statica) se si limitano a reperire risorse già presenti nel web server, **dinamiche** (con URL dinamica) se la risorsa richiesta viene determinata sul momento da una web application.

Jakarta EE

Jakarta EE (*Enterprice Edition*) è un insieme di specifiche che estendono la JDK standard per supportare lo sviluppo di applicazioni commerciali.

A noi interessa Jakarta Web Profile, che comprende:

- JSP : Jakarta Server Pages, un misto di html e Java (non ne faremo uso);
- Servlet: più flessibili, specie se le risorse richieste richiedono significative elaborazioni di dati: possono calcolare qualsiasi tipo di risorsa, non solo HTML.

Nello specifico, creeremo classi che estenderanno la classe HttpServlet e faranno override di alcuni suoi metodi per gestire richieste http di vari tipo (GET, POST, …).

Sarà il web container a gestire il ciclo di vita delle servlet.

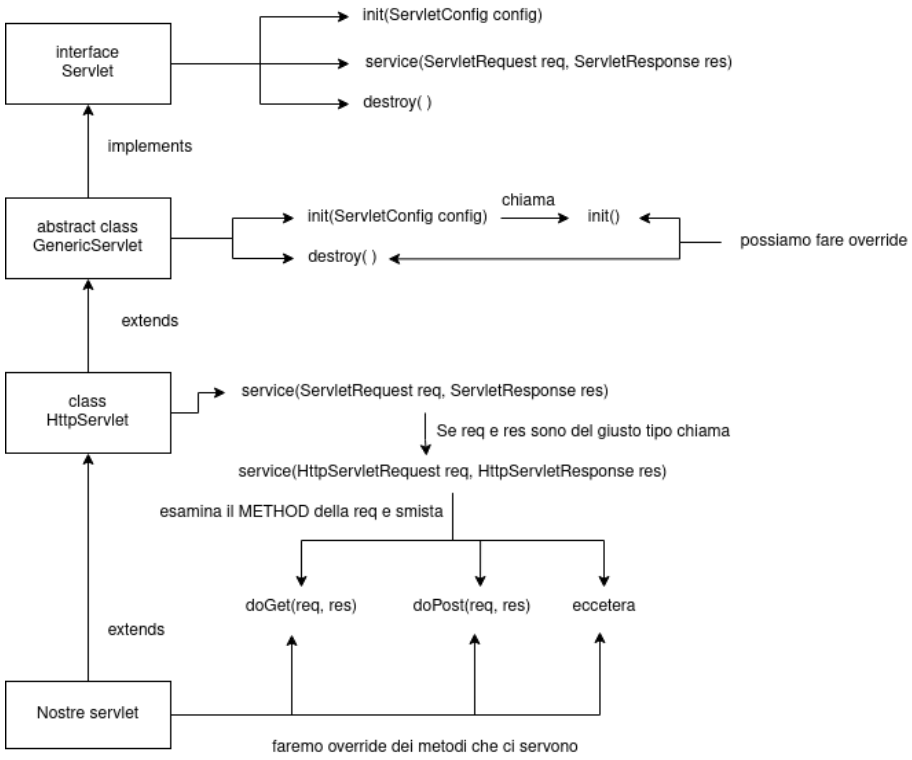
Quello che faremo sarà sostanzialmente questo:

- **init**: oltre ai sorgenti, un'applicazione di questo tipo prevede tipicamente dei file di configurazione utilizzati nelle fasi successive. Tipicamente viene gestita tramite tool di automazione che automatizzano diversi aspetti dello sviluppo e della costruzione dell'applicazione stessa. L'inizializzazione, sempre gestita dai tool di automazione, stabilisce una configurazione di partenza che potrà poi essere modificata nelle fasi seguenti;
- **develop**: progettazione e implementazione del codice, compilazione ed esecuzione, testing e debugging (tipicamente con librerie specifiche di supporto, es. Junit). Per tutte queste attività il programmatore è supportato dal tool di automazione, inoltre può avvalersi di un IDE (ambiente integrato di sviluppo) che facilita l'integrazione dei diversi strumenti utilizzati;
- **build**: il codice viene compilato definitivamente, viene "impacchettato" (packaged) in un formato adeguato alla fase successiva, che non prevede l'accesso o la modifica del codice stesso (ad esempio in Java"impacchetterò" soltanto i file .class). Tipicamente il pacchetto contiene anche file di configurazione necessari all'esecuzione nel Web Server;
- **deploy**: l codice viene installato sul server; server diversi offrono contesti di esecuzione diversi, e a maggior ragione diversi dal contesto di esecuzione in fase di sviluppo. Questo step può richiedere qualche ulteriore configurazione per permettere l'esecuzione nel contesto di esecuzione che è target (obiettivo) finale del deployment. Esistono tool con il preciso scopo di facilitare il deploy delle applicazioni su Web Server diversi (ma non li tratteremo in questo corso) - es.: Cargo, Docker, Kubernetes.

Servlet

In basso c’è uno schema riguardante i metodi che ci interessano e la gerarchia delle classi.

Vediamo il ciclo di vita delle servlet:



- **Start**: il Web Container crea un ServletContext comune a tutte le servlet; per ogni servlet *s* viene creata un'istanza e un'oggetto ServletConfig *sc* (tra le altre cose contiene un riferimento al ServletContext), e viene chiamato *s.init(sc)*. Questo metodo chiama *init()*, di cui possiamo fare override se abbiamo necessità. A questo punto la servlet è attiva. Il Web Cointainer tiene un "indirizzario" le servlet e le route a cui rispondono;
- **Request loop**: si divide in tre fasi.
  1) *matching & routing*: il Web Container instrada le richieste alle servlet in base alla URL ricevuta e agli urlPatterns dichiarati dalla servlet. Il pattern che più lungo che fa match "vince". Altrimenti *404 not found*;
  2) *predisposizione al servizio*: vengono create la request (HttpServletRequest), che contiene le informazioni sulla richiesta http arrivata al web server, e la response (HttpServletResponse), che conterrà la risposta http elaborata dalla servlet;[1]
  3) *service*: il Web Container chiama sulla servlet *s* vincente *s.service(request, response)*. L'implementazione di HttpServlet si occuperà di chiamare il metodo opportuno (doGet(), …) (vedi schema).
- **Stop**: per ogni servlet *s* viene chiamato *s.destroy()*. Come scritto nello schema di sopra, *destroy()* non fa nulla a meno che non facciamo *override*. Può essere utile per liberare risorse occupate nella *init()*.

1 Tecnicamente response è di tipo (class) *HttpServletResponseWrapper* implements → (interface) *HttpServletResponse* extends → (interface) *ServletResponse*. Stesso discorso per request.

Attenzione alla concorrenza

I metodi doXXX vengono eseguiti in thread, è possibile che ci siano due thread che eseguono lo stesso metodo doXXX.

JDBC

Usiamo i DB per garantire la persistenza dei dati nella nostra applicazione web. Ma come interagiamo ci interagiamo da java? Esiste il JDBC (Java DataBase Connectivity), un insieme di classi e interfacce a cui i driver dei DB si devono conformare. Basterà installare il driver giusto.

Grazie al JDBC il codice Java è indipendente dal DB utilizzato.

Useremo tre classi/interfacce:

- Connection: la connessione al db;
- Statement: la richiesta al db. Può essere una singola query in una stringa oppure una query parametrica (“*select * from tizio where id = ?”*) in cui possiamo specificare in seguito il valore del segnaposto “?” (*PreparedStatement*);
- ResultSet: il risultato di una *select*, posso leggere i valori delle colonne nella riga corrente o spostare il puntatore a una riga successiva (per leggere un’altra tupla).

Ci sono due modi per ottenere un oggetto *Connection*:

- *Driver.getConnection(url, username, password)*: crea una sola connessione;
- *dataSource.getConnection()*: un oggetto DataSource che fa uso di connessioni riusabili di un Connection Pool (bisogna includere una libreria esterna di Connection Pooling). Deve essere inizializzato solo una volta.

Creare e chiudere DataSource facilmente

Per farlo bisogna usare la programmazione ad eventi (con qualche differenza rispetto a quella spiegata a prog3).

Possiamo registrare un listener per eseguire operazione al momento della creazione e chiusura del ServletContext. Basta usare la Java Annotation e scrivere “*@WebListener*” prima della definizione di una classe che estende *ServletContextListener* (e ovviamente implementare i metodi necessari).

JSON

JSON (JavaScript Object Notation) è un formato di rappresentazione testuale di dati strutturati nato in ambiente JavaScript.

Una stringa JSON può rappresentare un oggetto, come un insieme di coppie chiave-valore, o un array. I valori possono essere stringhe (delimitate da virgolette), numeri (interi o decimali), boolean, altri oggetti o array, oppure null.

Gli headers di request e response contengono il “content-type” che specifica il tipo dei dati scambiati. Per JSON è “application/json”. È possibile trasformare oggetti Java in JSON e viceversa usando la libreria GSON (by Google): una volta creato un oggetto Gson *gson* con *new Gson()* possiamo usare *gson.toJson(object)* e *gson.fromJson(JsonString, ClassName.class)*.

Inviare dati

Per inviare dati senza avere una classe Java di riferimento, possiamo decidere di:

- creare una classe ad hoc;
- usare una HashMap: in questo caso possiamo inviare solo coppie chiave-valore dello stesso tipo;
- usare oggetti JSON generici definiti nella libreria GSON: in questo caso il codice diventa più dipendente dalla libreria esterna.

Ricevere dati

Nel caso in cui non sappiamo che cosa sta inviando il client, possiamo usare anche qui gli oggetti menzionati al punto precedente. E cioè JsonElement, JsonObject, JsonArray, JsonNull e JsonPrimitive.

Sessione

Come sappiamo, *“http is a stateless protocoll”*, cioè ogni richiesta è indipendente dalle altre. La sessione viene costruita sopra http, usando i cookie. Questi contengono informazioni che verranno mandate al server ogni volta che il client si connette.

Usando il metodo *request.getSession()* possiamo ottenere un oggetto che implementa *HttpSession* (interface), che rappresenta la sessione corrente (o ne crea una nuova e un nuovo cookie di sessione). Con un oggetto *HttpSession* (che chiameremo *session*) possiamo, oltre a ottenere informazioni circa la sessione, salvare con *session.setAttribute(key, value)* altre informazioni che vogliamo mantenere tra una sessione e l’altra. Come ad esempio l’id dell’utente connesso. Ulteriori metodi li trovate qui.

Log in/log out

Un modo semplice di controllare il log in/log out è ottenere ogni volta un oggetto *HttpSession* per salvare e/o controllare un’informazione sull’utente (esempio, l’id corrispondente nel DB).

Altro modo più snello, è estendere la classe *HttpFilter* per creare, appunto, un nuovo filtro che viene chiamato prima dell’invocazione della servlet. Bisogna specificare su quali url si vuole applicare il filtro usando *@WebFilter(urlPatterns = “…”)*.

Javascript

Prototype

Un oggetto JavaScript è definito in base alle sue proprietà. Queste si distinguono in proprietà-valore e proprietà-funzione. Le prime sono assimilabili alle variabili di istanza degli oggetti Java, le seconde invece ai metodi.

In JavaScript l'ereditarietà si basa sul "prototype". Ogni oggetto ha un prototype, un altro oggetto da cui eredita le proprietà (di entrambi i tipi). Quindi più oggetti con stesso prototype condividono le stesse proprietà. La radice dell'ereditarietà è Object.prototype.

Particolarità:

- le proprietà possono venire aggiunte o rimosse a runtime da un prototype, e il cambiamento avviene lungo la "prototype chain"(tutti gli oggetti che ereditano dal prototype);
- quando si vuole accedere a una proprietà, la ricerca di questa avviene prima nell'oggetto, poi nel suo prototipo e così via lungo la catena dei prototipi. Se non viene trovata, viene restituito undefined.

Come creare un oggetto

Object literal

Al contrario di Java, possiamo creare un oggetto senza definirne la classe. Possiamo infatti definire direttamente le proprietà usando un array associativo:

```
let obj = {
      nome_e_cognome: "Pinco Pallino",
      esami: […],
      media: () => { … return media;}
}
```

In questo caso il prototype è Object.prototype.

Possiamo anche aggiungere e rimuovere proprietà in questo modo:

```
delete obj.nome_e_cognome; //rimuove la proprietà
Obj.nome = "Pinco"; //definisce una nuova proprietà
obj.cognome = "Pallino";
```

Per accedere ai valori possiamo usare la dot notation (obj.nome) o la square bracket notation (obj[nome]).

Classi

Le classi non sono altro che insiemi di oggetti inizializzati con uno stesso costruttore e con lo stesso prototipo. Esempio:

```
class X {
      constructor(nome,cognome){
            this.nome = nome;
            this.cognome = cognome;
```

```
	}

	print(){…}
}
```

Viene creato un prototipo X.prototype (che ha prototipo Object.prototype) che ha come proprietà il metodo print().

Quando usiamo

```
let obj =  new X(…)
```

viene creato un nuovo oggetto su cui viene eseguito il costruttore di X (il metodo constructor) e che ha per prototipo X.prototype.

Le proprietà assegnate dal costruttore (this.nome e this.cognome) appartengono soltanto al nuovo oggetto.

Quindi la prototype chain è:

Object.prototype → X.prototype → obj

Se invece definiamo una classe in questo modo

```
class Y extends X{…}
```

quando facciamo obj = new Y() la prototype chain è:

Object.prototype → X.prototype → Y.prototype → obj.

Get e Set

Si possono definire delle *proprietà calcolate*, cioè proprietà-funzione che all'esterno si comportano come proprietà-valore.

Esempio:

```
class X{

	constructor(nome){
		this._nome = nome; //per prassi si usa _nomeVar per indicare che
andrebbe trattata come si tratta una variabile private in java
	}

	get nome(){return this._nome;}
	set nome(nome){this._nome = nome;}
}
```

I metodi devono essere preceduto dalla parola chiave get/set. Ulteriori limitazioni nella struttura dei metodi sono:

- i get non hanno parametri e restituiscono un valore;
- i set hanno un solo parametro e non restituiscono nulla.

Le proprietà-get possono essere usate all'interno di un'espressione (console.log(obj.nome)), i set possono essere usati in un assegnamento (obj.nome = "Pinco").

Attenzione al this

Il this viene legato all'oggetto su cui il metodo viene chiamato. MA CIÒ VUOL DIRE CHE:

- se lo usiamo in un metodo non di classe, this fa riferimento alla finestra del browser;
- idem se è contenuto in un metodo passato come callback (il metodo viene scollegato dall'oggetto a cui appartiene).

Inoltre il this subisce la closure solo per quanto riguarda le arrow function; altrimenti viene scollegato e finisce col corrispondere alla finestra del browser.

Object.create

È possibile creare un oggetto usando Object.create(prototype), in tal caso l'oggetto creato appartiene alla classe Object ma ne possiamo specificare il prototipo:

- eredita dal prototipo proprietà e metodi, se ci sono;
- può essere usato per decorare un object literal con caratteristiche di un altro oggetto

Funzioni

Le funzioni sono oggetti. Al contrario di Java, non è necessario dare un valore a tutti i parametri: quelli non assegnati avranno valore undefined.

Inoltre, ogni funzione restituisce un valore: se non c'è una return esplicita, viene restituito undefined.

Come definirle

Possiamo definire una funzione in 4 modi.

Function

Innanzitutto possiamo usare la parola chiave "function", che può essere usata in due modi:

```
1) function nome_funzione(parametri ...){//codice}
2) let x = function(parametri ...){//codice}
```

Nel primo modo la funzione si dice "named", cioè ha un nome con cui può essere chiamata. Posso comunque assegnare la funzione a una variabile:

```
let y = nome_funzione;
```

Nel secondo modo la funzione si dice "unnamed", cioè di per sé non ha un nome, ma può essere chiamata usando il nome della variabile a cui viene assegnata.

Arrow function

In questo caso non si utilizza la parola chiave "function", e in ogni caso la funzione che viene creata è "unnamed".

Ci sono due modi per creare una arrow function:

```
1) let x = (parametri …) => {//codice}
2) let x = (parametri …) => //codice
```
Nel primo caso tra le graffe può esserci anche più di una istruzione; nel secondo, invece, può esserci una sola istruzione.

Closure

La closure è il meccanismo che determina le variabili visibili (e quindi usabili) da una funzione.
Nella closure distinguiamo tra “scope interno” e “scope esterno”.
Il primo è quello a cui siamo abituati con Java/C, cioè la funzione può usare i suoi parametri e le variabili che definisce nel suo corpo.
Il secondo, invece, dipende da dove si trova la funzione.
Se all’interno di un’altra funzione, allora può usare anche le variabili nello scope interno della funzione che la contiente. Per esempio

```
function x(){
      let z = 1;
      function y(){
            console.log(z);
            z++;
      }
      y();
}
```

In questo caso la funzione y può vedere (e quindi usare) la variabile z nello scope interno della funzione x;

Se x viene definita non è contenuta in nessun’altra funzione, allora il suo scope esterno “vede” tutte le variabili definite nello stesso file della funzione stessa. In questo caso si parla di scope globale.

Moduli

Piuttosto che avere in un solo script tutte le variabili/funzioni/classi/oggetti, possiamo definirle in script separati, mettendo in uno stesso script variabili/funzioni/classi/oggetti tra di loro correlati, e renderli disponibili in altri script usando le import e le export.

Un modulo è proprio uno script che fa uso di import/export.

Export

Ci sono due modi per esportare qualcosa da uno script ad un altro:

- standard export;
- default export.

Il primo si ottiene premettendo *export* alla dichiarazione di una variabile/funzione/classe/oggetto, ed è possibile esportare più variabili/funzioni/classi/oggetti.

Il secondo, invece, si usa quando si vuole esportare solo un cosa, e in questo caso bisogna premettere *export default* alla dichiarazione della cosa da esportare.

Import

In quanto azione complementare all'export, ci sono grosso modo due modi per importare qualcosa da un altro script. La struttura è simile: all'inizio dello script bisogna scrivere

```
import | from "path/to/file.js"
```

Cambia solo ciò che va scritto al posto di | :

- in caso di import da default export, posso usare un nome qualsiasi per riferirmi alla cosa importata. In questo caso sto facendo un ***default import***;
- posso scrivere {cosa1, cos2, …} in cui elenco il nome delle cose che sto importando, e con questo nome le userò nel mio script, ***standard import di base***;
- posso scrivere {cosa1 as th1, …} assegno un nuovo nome alle cose importate, ***standard import con renaming***;
- posso scrivere "* as nome", e ciò vuol dire che import tutto ciò che viene esportato da quel file, e gli aggiungo un prefisso (al posto di "nome" può andare qualsiasi cosa). In questo caso potrò accedere a quelle cose importate usando nome.x (dove x è il nome che l'elemento aveva nello script da cui è stato esportato), ***standard import con namespace***.

A che serve? Ecco il DOM

JavaScript viene usato per rendere interattive le pagine web. Come? Semplice: l'ambiente di esecuzione del browser mette a disposizione il DOM (Document Object Model).

Facciamo un passo alla volta.

Come includo javascript?

Basta mettere nell'header dell'HTML questo tag:

```
<script src="path/to/file.js" type="module"></script>
```

Come è fatto il DOM

Il DOM altro non è che una struttura ad albero così composta:

- *document*: un oggetto che funge da radice del DOM;
- *elementi*: nodi del DOM, ognuno dei quali rappresenta un elemento HTML (e quindi ha tag e eventuali attributi);
- *nodi testuali*: porzioni di testo dentro l'HTML.

Ci sono altri nodi che non abbiamo visto a lezione.

Che ce ne facciamo?

Possiamo utilizzare la radice document per attraversare il DOM, alla ricerca di altri elementi che vogliamo modificare, cancellare, aggiungere. Possiamo anche modificare l'attributo class per

modificare l’aspetto dell’elemento (usando i css) (Per vedere i metodi da chiamare guarda le slide della prof.ssa).

Programmazione ad eventi

Una volta ottenuto l’elemento che ci interessa dal DOM, possiamo registrare delle funzioni di callback, che verranno eseguite quando si quell’elemento si verificherà l’evento che vogliamo gestire.

Bubbling portami via

Attenzione al bubbling: gli eventi risalgono lungo il DOM, quindi è possibile che venga eseguita qualche callback di un altro elemento: per sicurezza inserire *event.stopPropagation()* quando non vogliamo che l’evento salga ulteriormente (dove event è il nome dell’evento).

Front end

Architetture web

Gli elementi principali della maggior parte delle applicazioni web sono:

- *user interface* che dialoga con l'utente e gli permette di interagire;
- *application business logic* in grado di ricevere le interazioni dell'utente ed elaborare i dati secondo una logica interna (cambia in base al settore di business in cui l'applicazione si va a collocare);
- *data persitence*: a fronte dell'elaborazione salvo dei dati in modo persistente per un uso successivo (DB);
- *servizi di terze parti* come ad esempio servizi di pagamento con carta di credito negli ecommerce.

Possono avvenire 5 tipi di flussi di informazioni:

- all'avvio dell'applicazione vengono prelevati i dati dalla data persistence, l'application business logic li elabora sfruttando o meno i servizi di terze parti (es. calcola costi di spostamenti in automobile per chiedere rimborso spese usando google per calcolare le distanze) e mostra qualcosa all'utente. La bussiness logic elabora sia i dati sia elementi di presentazione degli stessi (es. aggregare i dati per mese in cui è avvenuto il viaggio);
- l'utente agisce con l'interfaccia senza che ci sia un riflesso sui dati (es. espandere un box). Cambia solo il modo in cui l'utente guarda i dati;
- l'utente agisce con l'interfaccia e modifica i dati ma non in modo persistente (la business logic ha ripercussione sulla data persitence) (es. modifica che non deve ancora essere salvata in un google documenti);
- l'utente modifica in modo persistente i dati (es. salva un google documenti); l'evento utente passa dalla business logic che può decidere di trattare i dati prima di salvarli. A quel punto la business logic rilegge i dati nel complesso (es. salva un prodotto nel carello e rilegge i prodotti salvati nel carrello);
- un evento esterno può attuare un cambiamento dei dati che deve essere riflesso nell'interfaccia utente. In questo caso la sorgente dell'evento non è l'utente, ma viene dal server o dai servizi terzi (questa cosa non succederà nel nostro progetto).

Server-side vs client-side

Quanto discusso sopra, può essere diviso in vario modo tra sever-side e client-side. Per esempio possiamo avere applicazioni che si svolgono quasi interamente sul client e usano il server solo per salvare i dati (es. google document). Di solito sono applicazioni pensate per essere usate da un utente singolo. Può anche succedere che il grosso dell'application business logic risieda sul server e sul client c'è giusto un po' di interfaccia per permettere di interagire con i dati (es. moodle lato

studente). L’application business logic può anche essere divisa tra server e client (es. giochi online con regole del gioco sul server e renderizzazione 3D sul client).

Componenti

Un componente di una UI corrisponde a un’area precisa sullo schermo, tipicamente a un elemento del DOM con il suo sottoalbero. Un componente può contenere uno o più componenti, in tal caso il primo è detto componente-genitore (*parent*) e gli altri componenti-figli (*child components*), che sono detti “innestati” (*nested*) nel genitore. Il genitore è responsabile della configurazione e della comunicazione tra i figli.

Quasi sempre, il componente presenta un ben preciso aspetto del modello dati (a cui fa riferimento la *business logic*). Lo visualizza, talvolta permette di modificarne alcune caratteristiche. Ci sono anche i casi in cui un componente ha soltanto il ruolo di facilitare l'interazione, senza necessariamente riferirsi ai dati su cui l'applicazione lavora: es una barra di scorrimento, un box di conferma.

Single Page Application

La gerarchia dei componenti può essere divisa in due categorie:

- **statica**: determinata da tutti i possibili sotto-componenti che ciascun componente può contenere e visualizzare nel corso del suo ciclo di vita;
- **dinamica** all'istante T: data da tutti i sotto-componenti che ciascun componente contiene e visualizza nell'istante T.

Questo perché il DOM può essere modificato a runtime.

Questo vuole dire che il componente-radice (App in React) può avere nella gerarchia statica dei componenti che corrispondono a pagine diverse e mostrarne uno alla volta. In questo caso di parla di **Single Page Application**, perché tutte le schermate possono essere mostrate su una sola pagina (la radice ha più *viste*).

Model-View-ViewModel

Questo pattern architetturale prevede tre parti:

- model: i dati che l'app elabora, i vincoli che ne regolano la fruizione e l'elaborazione;
- view: ciò che l'utente vede e con cui interagisce (il DOM);
- view-model: gestisce il comportamento dei componenti (modificando quindi il DOM). Gestisce ogni interazione tra model e view.

Binding

Nei framework & affini il collegamento fra view-model e view è parzialmente automatizzato. Ogni framework (o affine) adotta la sua strategia per accorgersi di quando qualcosa "cambia" nel view-model e aggiornare di conseguenza la view.