TECNOLOGIE AVANZATE PER FRONTEND

REACT.JS

React è una **libreria JS** per la creazione di interfacce utente (GUI) web. L'obiettivo è diventare la soluzione semplice, intuitiva e definitiva per sviluppatori front-end e app basate su HTML5.
Essendo una libreria JS viene eseguito all'interno del browser ne consegue che React.js NON è uno strumento per lo sviluppo lato back-end. Quindi React non interagisce con DB o qualsiasi altra sorgente di dati si trovi su back-end, tuttavia permette di invocare delle API lato server. Può interagire con diverse tecnologie back-end in Python/Flask, Ruby on Rails, Java/Sping, PHP ecc

L'approccio in sintesi

React si ispira alla metodologia di sviluppo delle GUI del tipo Single Page Application (SPA). Una **SPA** è un'applicazione web che interagisce col browser per modificare pagine web in modo dinamico dei in funzione dei dati che arrivano dal back-end, a differenza dell'approccio classico in cui il browser carica nuove pagine a seguito dell'interazione con l'utente. Si dice, infatti, che la SPA è un contenitore all'interno del quale la pagina evolve dinamicamente.
Lo sviluppo dell'applicazione avviene attraverso la scrittura di "componenti" i quali interagiscono con le API della libreria che manipolano, a loro volta, il DOM per la creazione di elementi di interfaccia utente.

React ha introdotto il concetto di Virtual DOM: al verificarsi di un evento, invece di modificare il DOM del browser, modifica una esatta copia del DOM (il virtual DOM) del browser e si trova in RAM. La modifica del virtual DOM è più leggera di quella del DOM browser, infatti lavorando su di esso React sarà in grado di inviare al DOM del browser solo le modifiche strettamente necessarie rendendo il processo di rendering della pagina più leggero, efficiente e veloce.

Ma quindi cosa succedeva con il DOM normale?

> Se si modificasse un solo elemento del DOM (supponendo di averne tanti), con la tecnologia normale si andrebbe a riscrivere l'intero DOM

E React cosa fa?

> React dice al browser di modificare solo le parti effettivamente modificate e possibilmente, prima di modificare il DOM del browser, accorpa un certo numero di modifiche fatte dall'utente e decide lui quando mandare le modifiche al browser

I vantaggi

I vantaggi per lo sviluppatori sono:

- L'approccio a componenti permette allo sviluppatore di costruire interfacce complesse attraverso la composizione di mattoncini
- Un altro vantaggio dell'approccio a componenti è la possibilità di riuso in modo semplice
- Lo sviluppatore definisce la logica dei componenti e la loro posizione all'interno della GUI. La gestione del virtual DOM, delle sue trasformazioni e della comunicazione con il DOM del browser è completamente a carico di React.js

Per l'utente finale invece l'impiego del virtual DOM alleggerisce il processo di rendering dell'interfaccia sul browser con conseguente aumento delle prestazioni percettibili dall'utilizzatore.

Un primo approccio con React può essere:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <title>Primi passi con React</title>
        <!-- IMPORT DELLE LIBRERIE REACT -->
        <script src="https://unpkg.com/react@15/dist/react.js"></script>
        <script src="https://unpkg.com/react-dom@15/dist/react-dom.js"></script>
        <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.24/browser.js"></script>
    </head>
    <body>

        <!-- CREO IL CONTENITORE -->
        <div id="root"></div>

        <script type="text/babel">
            //  CREAZIONE DI UN REACT ELEMENT
            const elem = <p>Hello <strong>React</strong>!</p>;

            //  REINDIRIZZA IL TUTTO:
            //      - elem è cosa visualizzare
            //      - il document.getElementById(...) è dove visualizzarlo
            ReactDOM.render(elem, document.getElementById('root'));
        </script>
    </body>
</html>
```

Nell'esempio è stato definito un *React Element* e successivamente è stato chiesto al DOM del browser di visualizzare l'elemento in una specifica posizione. Un **React Element** è un oggetto semplice ed immutabile che descrive cosa si vuole visualizzare sullo schermo. Solitamente un element è un nodo html ma può anche avere al suo interno istanze di componenti.
La notazione `const elem = <p>Hello <strong>React</strong>!</p>;` è un esempio di JSX (JavaScript XML), difatti in JS vanilla non si può fare.

Un altro esempio:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <title>Primi passi con React</title>
        <script
        src="https://unpkg.com/react@15/dist/react.js"></script>
        <script src="https://unpkg.com/react-dom@15/dist/reactdom.js"></script>
        <script src="https://cdnjs.cloudflare.com/ajax/libs/babelcore/5.8.24/browser.js"></script>
    </head>
    <body>
        <div id="root"></div>
        <script type="text/babel">
            // SI POSSONO DEFINIRE DEI COMPONENT DI TIPO CLASS
            class HelloWorld extends React.Component{
                render() { // Funzione che ritorna l'elemento da visualizzare
```

```
                return <p>Hello <strong>React</strong>!</p>;
            }
        };
        ReactDOM.render(<HelloWorld />,document.getElementById('root'));
    </script>
  </body>
</html>
```

Nell'esempio precendente il risultato è lo stesso di quello che lo precede ma è fatto tramite una classe.
Si noti che quando vado a fare il `ReactDOM.render(...)` metto un tag "nuovo" che ha lo stesso nome della classe.
La differenza sostanziale fra i due esempi sta nel fatto che nel primo ho usato un **React Element** mentre nel secondo ho usato un **React Component**. I *React component* possono essere di tipo class (come nell'esempio) o di tipo function. Questi componenti sono oggetti complessi e dinamici che ricevono input dall'esterno e forgiano l'elemento grafico da restituire. Sostanzialmente ricevono dei dati in ingresso come delle funzioni e restituiscono l'elemento grafico da visualizzare.

JSX

Negli esempi visualizzati è stata usata una sintassi particolare che mescola JS e HTML: **JSX**. JSX sta per JavaScript XML e permette allo sviluppatore di scrivere facilmente tag HTML all'interno di codice JS e di piazzarli all'interno del DOM senza l'uso di `createElement()` e/o `appendChild()`.
Non è obbligatorio usare JSX però semplifica molto la vita del developer. React però mette comunque a disposizione funzioni per la creazione di elementi HTML.
Un esempio dell'utilità di JSX (creazione di una lista di 3 elementi):

- Con JSX:

```
// Creazione della lista degli elementi -->
const listElement = <ul className="list-of-items">
        <li className="item-1" key="key-1">Item 1</li>
        <li className="item-2" key="key-2">Item 2</li>
        <li className="item-3" key="key-3">Item 3</li>
    </ul>;
// Esecuzione del rendering nella pagina
ReactDOM.render(listElement, document.getElementById("container"));
```

- Senza JSX:

```
// Creazione degli elementi da inserire in una lista non ordinata
var item1 = React.DOM.li({ className: "item-1", key: "key-1"}, "Item 1");
var item2 = React.DOM.li({ className: "item-2", key: "key-2"}, "Item 2");
var item3 = React.DOM.li({ className: "item-3", key: "key-3"}, "Item 3");
// Creazione di un array degli elementi
var itemArray = [item1, item2, item3];
// Creazione della lista degli elementi
var listElement = React.DOM.ul({ className: "list-of-items" }, itemArray);
// Avvio del rendering nella pagina
ReactDOM.render(listElement, document.getElementById("container"));
```

Si possono inserire delle variabili all'interno della notazione JSX come segue:

```
const nome = 'Giuseppe Verdi';
const element = <h1>Hello, {nome}</h1>;
ReactDOM.render(element, document.getElementById('root'));
```

Per inserire le variabili, espressioni di ogni tipo e funzioni che restituiscono valori si mettono fra parentesi graffe `{}`.
Ma come fa a funzionare?

> Funziona perchè prima di essere interpretato dal browser, il codice che include JSX, viene pre-compilato da un interprete che è in grado di tradurre il codice JSX in JavaScript. In questo corso si usa **babel** (open source). Babel mette a disposizione tool per tradurre molti linguaggi in JS.

A runtime sostanzialmente viene eseguito un compilato che corrisponde al javascript che è stato dedotto dal codice scritto in React.

I COMPONENTI

I **React Components** sono i mattoncini fondamentali che consentono di passare da una pagina statica a un'applicazione web dinamica la cui interfaccia è in grado di ripondere agli eventi che si verificano nella pagina, ossia reagire (e da qui il nome React) e aggiornare se stessa di conseguenza. Ognuno di questi ha un ruolo ben definito dal punto di vista di ciò che rappresenta graficamente e si fa carico di gestire le interazioni con l'utente su quella particolare interfaccia.

- La sezione più esterna, quella col bordo giallo, è il componente React che rappresenta l'applicazione e contiene tutti gli altri componenti.
- A "livello" più basso, il riquadro blu contiene il pannello per la ricerca incrementale dei prodotti
- Allo stesso livello, con colore verde, c'è la lista dei prodotti che a sua volta è formata da altri componenti interni
- Per ogni riquadro di colore diverso sarà stato dichiarato un componente React

Tipi di componenti

In React i componenti sono pezzi di codice indipendenti e riusabili. Esistono 2 tipi di componenti:

- il tipo **class**
- il tipo **function**

Entrambe devono reindirizzare del codice HTML. La cosa che li differenzia è che quelli di tipo function non salvano lo stato, se si necessita di salvare lo stato bisogna usare i componenti di tipo class.

Regola generale per la definizione di un componente è che il nome del componente deve avere la lettera maiuscola.
Per definire un componente di tipo function:

```
function Car() {
    return <h2>I am a Car!</h2>;
}
ReactDOM.render(<Car />, document.getElementById('root'));
```

La funzione deve restituire l'elemento di cui fare il rendering tramite `return`. Come già visto `ReactDOM.render(...)` è l'istruzione che attiva la manipolazione del DOM e il successivo rendering del browser.

Per definire un componente (uguale) di tipo class:

```
class Car extends React.Component {
    render() {
        return <h2>Hi, I am a Car!</h2>;
    }
}
ReactDOM.render(<Car />, document.getElementById('root'));
```

Per creare un componente di tipo class, occorre creare una classe che estenda da React.Component e implementi obbligatoriamente il metodo `render()`. Così come per i componenti di tipo function, occorre che questo metodo restituisca l'elemento da renderizzare attraverso la parola chiave `return`. Come si vede il metodo di visualizzazione sul DOM è uguale.

Sia per i componenti di tipo class che quelli di tipo function si possono definire delle **props** (proprietà). Sono dei parametri in sola lettura che si passano all'oggetto, sono immutabili e sono utili per configurare, per esempio, il comportamento grafico del componente. L'oggetto built-in che contiene queste proprietà prende il nome di props (keyword riservata). Quando si fa il rendering si può accedere alle props di un component richiamandole come se fossero attributi di un tag HTML.

```
function Car(props) {
    //props.colore è READ ONLY per Car
    return <h2>I am a {props.colore} Car!</h2>;
}
ReactDOM.render(<Car colore="red"/> /*qua colore lo posso cambiare*/,
    document.getElementById('root'));

class Car extends React.Component {
    render() {
        //Per le class si usa this.props
        return <h2>Hi, I am a Car. My name is {this.props.nome}</h2>;
    }
}
ReactDOM.render(<Car nome="Saetta McQueen"/>, document.getElementById('root'));
```

Essendo un oggetto JS che punta ad un'area di memoria che può non contenere dei campi con determinati nomi se non sono stati definiti nel rendering (si rischia che sia un puntatore a nullo).
Ogni volta che si mette un tag si crea un oggetto diverso quindi si possono definire diversi valori delle props. Ma perchè questa scelta?

> Se ho tanti componenti il passaggio per valore (e non per indirizzo) è utile sapere che se le proprietà cambiano inaspettatamente non è perchè è scritto male il componente.

Eventualmente si può utilizzare una factory per la creazione inline delle classi:

```
var Car = React.createClass({
    render: function() {
        return <h2>Hi, I am a Car!</h2>;
    }
});
```

Il concetto di state

Tutti i componenti di tipo **class** possiedono un oggetto built-in che prede il nome di **state**. A differenza delle *props* le proprietà definite nell'oggetto state SONO mutabili, infatti state è pensato proprio per contenere proprietà che possono cambiare nel tempo. Quando si cambia un attributo all'interno di state viene invocata la ri-renderizzazione del relativo componente.
N.B.: le componenti **function** sono stateless, ovvero l'oggetto state non lo possiedono.
Come per tutti i linguaggi ad oggetti anche per il tipo class si può definire un costruttore che viene invocato prima del rendering e funge da inizializzatore delle proprietà del componente. Il costruttore in generale serve per inizializzare lo stato del componente e per inizializzare la gestione degli eventi.
Nel costruttore si può invocare il metodo `super()` per invocare il costruttore dell'oggetto padre e per inizializzare correttamente il componente stesso. Se non si usa non si potrà utilizzare la keyword `this`.

```
class Car extends React.Component {
    constructor() {
        super();
        this.state = {brand: "Ford", model: "Mustang", color: "red", year: 1964};
    }
    render() {
        return (
            <div>
            <h1>My {this.state.brand}</h1>
            <p> It is a {this.state.color} {this.state.model} from {this.state.year}</p>
            </div>
        );
    }
}
ReactDOM.render(<Car />, document.getElementById('root'));
```

Si possono includere anche componenti in altri componenti, e quando verrà fatto il render del componente che ne contiene altri, viene fatto anche di tutti i sotto componenti.

L'oggetto state di un componente class può essere modificato attraverso la funzione `setState()` che viene definita nella classe `React.Component` e quindi viene ereditata dai componenti class. L'invocazione di tale funzione scatena la re-invocazione della funzione `render()` del componente e di tutti i suoi componenti nested. L'oggetto state è incapsulato all'interno di un componente, il quale è l'unico ad avere diritto e responsabilità di mutarlo, nessun altro può modificarlo: ciò è utile per quando si debugga, sapendo che lo stato è stato modificato in modo errato allora l'errore sta per forza nel componente a cui appartiene lo stato sbagliato.

Esempio di un componente per il lancio di un dado:

```
class Dado extends React.Component {
    constructor(props) {
        super(props);
        this.state = {numeroEstratto: 0};
    }
    randomNumber() {
        return Math.round(Math.random() * 5) + 1;
    }
    lanciaDado() {
        this.setState({numeroEstratto: this.randomNumber()});
    }
    render() {
        let valore;
        if (this.state.numeroEstratto === 0) {
            valore = <small>Lancia il dado cliccando <br /> sul pulsante
            ottostante</small>;
        }else{
            valore = <span>{this.state.numeroEstratto}</span>;
        }
        return (
            <div className="card" >
                <p className="card__number">{valore}</p>
                <button className="card__button" onClick={() => this.lanciaDado()}>
                    Lancia il Dado
                <button>
            </div>
        )
    }
}
```

Nell'esempio, per farlo funzionare correttamente, bisogna inserire tutto il resto:

- Tag html, head, body
- Inclusione delle librerie di React e JSX
- Rendering del componente all'interno di un div contenitore

Uso raccomandato di state e props Non tutti i componenti dovranno avere state, al contrario è consigliato costruire componenti **stateless**. Solitamente l'applicazione React è realizzata come una gerarchia di componenti: ci sono componenti ai vertici che saranno resposabili di mantenere lo stato dell'applicazione e di passare le informazioni giù ai componenti figli tramite *props*.

GESTIONE DEGLI EVENTI

Gli eventi sono solitamente gestiti da un **handler** realizzato attraverso un metodo della classe.
Facciamo subito un esempio pratico:

```
class App extends React.Component {
    constructor(props) {
        super(props);
    }
```

```
    /*HANDLER DELL'EVENTO*/
    handleClick(e) {
        console.log("Pulsante premuto - Evento click");
    }
    render() {
        return (
            <button onClick={this.handleClick} >Pulsante</button>
        )
    }
}
```

Il parametro *e* (nella firma dell'handler) è un evento sintetico, React definisce questo tipo di eventi in base alle specifiche W3C, quindi non ci sono problemi di compatibilità tra browser.
È utile sapere che gli eventi React sono lievemente diversi rispetto agli eventi nativi di JS.
N.B.: se l'handler dell'evento deve fare accesso allo *state* del componente occorre apportare accorgimenti al codice di gestione dell'evento.

Per accedere allo state da parte dei metodi di classe innanzitutto è necessario che l'oggetto che invoca l'handler dell'evento sia il componente: a tal fine si ricorrerà alla keyword *this*. Ci sono 2 alternative:

- All'interno del costruttore, forzare bind di this del metodo a this del componente.
  Ad esempio:

```
class Interruttore extends React.Component {
    constructor(props) {
        super(props);
        this.state = {acceso: true};
        this.handleClick =
            this.handleClick.bind(this);
    }
    handleClick() {
        this.setState({acceso: !this.state.acceso}
        // in alternativa
        // this.setState(state => ({
        // acceso: !state.acceso
        // }));
    }
    render() {
        return (
            <button onClick={this.handleClick}>
                {this.state.acceso ? 'Acceso' : 'Spento'}
            </button>
        );
    }
}
```

- Invocare l'handler come arrow function.
  Ad esempio:

```
class Interruttore extends
    React.Component {
        constructor(props) {
            super(props);
            this.state = {acceso: true};
        }
        handleClick() {
            this.setState({acceso:
            !this.state.acceso})
            // in alternativa
            // this.setState(state => ({
            // acceso: !state.acceso
            // }));
        }
        render() {
            return (
                <button onClick = {() = >
                    this.handleClick()}>
                    {this.state.acceso ? 'Acceso' : 'Spento'}
                </button>
            );
        }
    }
```

I FORM

È utile sottolineare che gli elementi dei **form** in React funzionano in modo leggermente differente rispetto ad HTML, e la motivazione fondamentale è che gli elementi di un form mantengono, naturlamente, uno stato interno. Infatti gli elementi di un form (`<input>`, `<texarea>`, ...) mantengono e aggiornano il proprio stato in base all'input dell'utente.
Nel seguente esempio si creerà un input in react:

```
class EsempioForm 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});
        console.log('onChange: lo stato ora vale ' +
        event.target.value);
    }
    handleSubmit(event) {
        alert('E\' stato inserito un nome: ' + this.state.value);
        //previene l'esecuzione del comportamento predefinito
        event.preventDefault();
    }
```

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

Invocazione risorsa sul server In React si possono effettuare HTTP request in diversi modi: uno dei più eleganti e semplici fa uso di *Fetch API* fornite da JS nativo. Esse forniscono un'interfaccia JS per accedere e manipolare parti della pipeline HTTP (request e response), inoltre mettono a disposizione un metodo che fornisce un modo semplice e logico per recuperare le risorse in modo asincrono. Nell'esempio seguente viene mostrata la composizione di una request HTTP di tipo POST all'interno di un handler (`FormData` è un'interfaccia JS nativa supportata da tutti i browser):

```
class MyForm extends
React.Component {
    constructor() {
        super();
        this.handleSubmit =
        this.handleSubmit.bind(this);
    }
    handleSubmit(event) {
        event.preventDefault();
        const data = new FormData(event.target);
        fetch('/api/form-submit-url', {
            method: 'POST',
            body: data,
        });
    }
    render() {
        return (
            <form onSubmit={this.handleSubmit}>
                <label htmlFor="username">Enter username</label>
                    <input id="username" name="username" type="text" />
                <label htmlFor="email">Enter your email</label>
                    <input id="email" name="email" type="email" />
                <label htmlFor="birthdate">Enter your birth date</label>
                    <input id="birthdate" name="birthdate" type="text" />
                <button>Send data!</button>
            </form>
        );
    }
}
```

LIBRERIE E FRAMEWORK ALTERNATIVI A REACT.JS

Oltre a React esistono numerose iniziative che propongono librerie e framework basate su javascript. L'obiettivo di ciascuna iniziativa è quello di fornire allo sviluppatore uno strumento/ambiente di sviluppo lato front-end più "comodo" rispetto a javascript (soprattutto per la gestione del DOM) e che possa abbattere i tempi di sviluppo delle interfacce delle applicazioni Web. Di seguito proponiamo alcune tra librerie/framework più popolari e più utilizzati, elencandone le caratteristiche principali.

JQUERY La libreria opensource jQuery è in assoluto la più utilizzata e conosciuta dalla comunità degli sviluppatori. JQuery semplifica molto la gestione degli elementi DOM e presenta diverse funzioni per questo scopo: con i selettori del CSS3 si possono selezionare facilmente e manipolare gli elementi della pagina. Inoltre, offre una gestione semplificata delle richieste Ajax. Il codice è compatibile con tutti i browser ed esistono molti plug-in. È una componente essenziale di molti CMS come WordPress, Drupal o Joomla! La sua estension jQuery UI è particolarmente adatta per realizzare effetti semplici ed elementi interattivi come drag&drop, ingrandimento e ridimensionamento degli elementi del sito, animazioni ed effetti vari.

Angular Creato e mantenuto da Google, è il successore di AngularJS. Insieme a React.js, dispone di una grande community di sviluppatori. È riconosciuto come l'antagonista principale di React.js. Analogamente a React.js, serve per realizzare Single Page Application. Implementa il design pattern MVVM (Model View ViewModel). Si basa su jQuery Lite, una variante compatta della altrettanto famosa libreria js jQuery. Rispetto al suo antecedente (AngularJS) la differenza principale è che per la programmazione non viene più utilizzato JavaScript, ma TypeScript, un linguaggio di programmazione sviluppato da Microsoft che si basa su javascript. Punto di forza è la facilità di sviluppo delle applicazioni per diversi dispositivi (desktop, mobile, tablet).

VUE.JS Analogamente ad Angular e React, Vue.js è un framework js per lo sviluppo di Single Page Application. Adotta il design pattern Model–View–ViewModel. L'intento degli sviluppatori di Vue.js è stato quello di creare uno strumento più facile per i principianti rispetto agli altri framework. Ciò, però, va a discapito della completezza di funzionalità (in cui i competitor eccellono), per le quali però è comunque possibile integrare un numero ristretto di librerie aggiuntive opzionali.

METEOR Meteor, chiamato a volte MeteorJS, è un framework javascript particolarmente adatto per lo sviluppo su diverse piattaforme. Consente agli sviluppatori di creare con lo stesso codice sia applicazioni Web sia app per i dispositivi mobili. Un altro vantaggio consiste nel fatto che le modifiche al codice possono essere inoltrate direttamente ai client grazie al protocollo proprietario Distributed Data Protocol (DDP). Questo framework js funziona su una base Node.js (ne parleremo presto), pertanto può essere impiegato sia per sviluppo front-end che per sviluppo back-end Risulta molto utile disporre di conoscenze su Node.js. per lavorare con Meteor.

BACKBONES Backbones non è un vero e proprio framework ma, piuttosto, un ottimo strumento per modellare e strutturare il codice. Grazie a questa caratteristica, backbones lascia più spazio al programmatore. Per contro, impiegato da solo non fornisce un framework completo, quindi lo si deve abbinare obbligatoriamente ad altre librerie quali underscore.js e jquery. È nato per sviluppare applicazioni single-page ed adotta il design pattern Model-View-Presenter (MVP).

/newpage

TECNOLOGIE AVANZATE PER BACKEND

JAVA MODEL 2

Nel progetto di applicazioni web in Java esistono 2 modelli di ampio uso e riferimento: **Model 1** e **Model 2**.
Per quanto riguarda **Model 1** si può dire che è un pattern semplice in cui codice il codice responsabile per presentazione contenuti è mescolato alla logica di business (suggerito solo per applicazioni piccole).
Model 2 è design pattern più complesso e articolato che separa chiaramente il livello di presentazione dei contenuti dalla logica utilizzata per manipolare e processare contenuti stessi (suggerito per applicazioni medio-grandi). Usualmente questo design pattern è associato al paradigma MVC (Model View Controller).

ARCHITETTURA MVC

Architettura adatta per le web app interattive (ma non solo). È composto da:

- **Model**: rappresenta il livello dei dati, incluse le operazione per accesso e modifica. Model deve notificare view associate quando modello viene modificato e deve supportare:
  - possibilità per view di interrogare lo stato di model
  - Possibilità per controller di accedere alle funzionalità incapsulate da model
- **View**: si occupa del rendering dei contenuti di model. Accede ai dati tramite model e specifica come i dati debbano essere presentati:
  - aggiorna presentazione dei dati quando model cambia
  - gira input utente verso controller
- **Controller**: definisce comportamento dell'applicazione (contiene la logica di business):
  - fa dispatching di richieste utente e seleziona view per presentazione
  - interpreta input utente e lo mappa su azioni che devono essere eseguite dal model

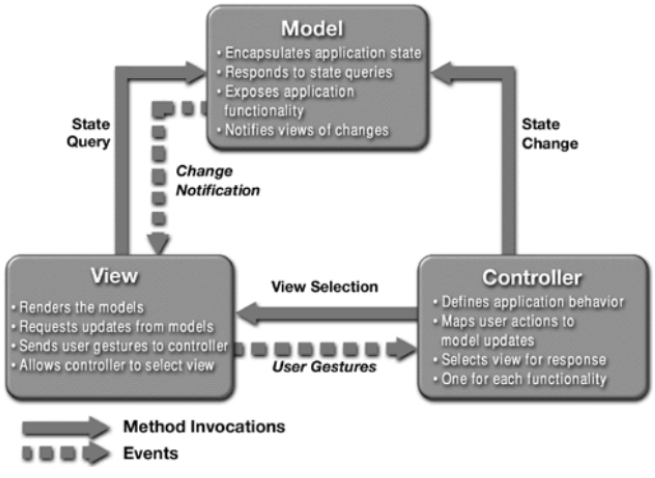

Figure 1: Model View Controller