Javascript - Temi Trasversali (III parte)

Type Checking e Typescript

Problema della tipizzazione dinamica in JavaScript

- JavaScript è un linguaggio con tipizzazione dinamica: il controllo dei tipi avviene a runtime.
- Questo rende difficile il debugging e l'analisi statica del codice.
- Il comportamento del codice può variare in base ai tipi dei parametri passati alle funzioni.
- La validazione del codice richiede l'esecuzione in tutti i possibili scenari, diventando insostenibile.

Typescript come soluzione

- Typescript è un linguaggio open source sviluppato da Microsoft (dal 2012).
- È un superset di JavaScript: tutto il codice JavaScript valido è anche codice Typescript valido.
- Aggiunge annotazioni di tipo e un sistema di type-inference per l'analisi statica.
- Un compilatore (transpiler) converte il codice Typescript in JavaScript, eseguendo il type checking.

Estensioni di Typescript

1. **Annotazioni di tipo:**
    - Implicite (per inferenza): `const el = 'Hello world';` // el è di tipo string
    - Esplicite: `const el: string = 'Hello world';`
2. **Dichiarazione di tipo in funzioni e array:**

```
function add(a: number, b: number): number {
  return a + b;
}

let arrayOfNumbers: number[] = [1, 2, 3, 4];
type arrayOfStrings = Array<string>;
```

3. **Tipo generico `any` :** Permette di assegnare qualsiasi tipo a una variabile.

```
 let abc: any = { a: 'x', b: 15, c: new Date() };
```

4. **Interfacce:** Definiscono la struttura di un oggetto.

```
interface Person {
  name: string;
  age: number;
}
```

5. **Classi:** Con modificatori di accesso `private` e `public` .

```
class Person {
  private codicefiscale: string;
  public name: string;
  public age: number;
```

```
    //...
}
```

6. **Estensioni di classi:** Ereditarietà e modificatori `static` e `readonly`.

```
class Employee extends Person {
  static numOfEmployee: number = 0;
  readonly cmp: string = "ACME";
    // ...
}
```

7. **Decoratori:** Funzioni che modificano classi, metodi, proprietà o parametri. Sono precedute da `@`.

```
class Person {
// ...
  @check
  toString(): string { ... }
}
```

Routing

Definizione

- Assegnazione di un URI diverso a ogni stato dell'applicazione web (selezione di contenuto).

Routing Server-Side

- Il browser si aspetta una pagina HTML completa come risposta.
- L'application logic decide quale contenuto inviare.
- Fattori decisionali:
  - Metodo HTTP (GET, POST, PUT, DELETE).
  - URI della richiesta (protocollo, dominio, path, query, fragment).
  - Header della richiesta (non di competenza del router).

Esempio con Express.js

- Libreria Node.js per il routing server-side.

```
// index.js
var express = require('express');
var app = express();
var routes = require('./routes.js');
app.use('/', routes);
app.listen(8000);

// routes.js
var express = require('express');
var router = express.Router();
router.get('/', function(req, res){ ... });
router.get('/about', function(req, res){ ... });
```

```
router.post('/contacts', function(req, res){ ... });
module.exports = router;
```

Routing Client-Side

- Non si basa sulla navigazione server-side.
- Utilizzato nelle Single Page Application (SPA).
- La pagina HTML è unica, l'URI non cambia mai.
- Problemi con navigazione, bookmark, back/forward, SEO, etc.

Approcci

1. **Client-Side Rendering (CSR):**
    - Pagina HTML iniziale vuota o con contenuto generico.
    - Il contenuto viene caricato dinamicamente tramite richieste XHR (AJAX).
    - Il DOM viene modificato lato client.
2. **Server-Side Rendering (SSR):**
    - Lo stato è mantenuto sul server.
    - Il server genera una nuova pagina HTML a ogni richiesta.
3. **Static Site Generators (SSG):**
    - Lo stato è mantenuto sul server.
    - Le pagine HTML vengono generate staticamente durante la compilazione.
    - Nessuna computazione durante la navigazione.

Gestione di `location` e `history`

- `window.location` : informazioni sull'URI corrente.
- `window.history` : stack degli URI visitati.
- È necessario gestirli esplicitamente per avere URI navigabili e utilizzabili.

Esempio di routing "fatto a mano"

```
// HTML
<nav>
  <ul>
    <li><a href="#" onclick="goto('home')">Home</a></li>
    </ul>
</nav>
<main id="content"></main>

// JavaScript
let routes = {
  home: `<h1>Welcome</h1>`,
  // ...
};

const content = document.getElementById('content');

function goto(destination) {
  content.innerHTML = routes[destination];
```

```
  let newUri = "/" + destination;
  window.history.pushState({ path: destination }, "", newUri);
}
```

Routing con framework

- **Angular:** `RouterModule`, `routerLink`.
- **React:** Libreria `react-router-dom`, componente `Route`.
- **Vue:** Libreria `vue-router`, componenti `RouterView` e `RouterLink`.

Routing Dinamico (o Parametrico)

- Gestione di parametri nell'URI.
- Esempi:
  - React: `<Route path="/users/:id" element={<User />} />`
  - Angular: `{ path: 'users/:id', component: UserComponent }`
  - Vue: `{ path: '/users/:id', component: User }`
  - Express: `app.get('/users/:id', function(req, res) { ... });`

Server-Side Generation

- **Next.js (React) e Nuxt (Vue):**
  - Pre-rendering di default.
  - Static Generation: pagine HTML generate a build time.
  - Server-Side Rendering: pagine HTML generate a ogni richiesta.

Binding

Definizione

- Collegamento tra dati usati in parti diverse di un programma.
- La modifica di un dato si propaga automaticamente agli altri dati collegati.
- Non confondere con il metodo `.bind()` delle funzioni JavaScript.

Tipi di Binding

- **Monodirezionale (one-way):**
  - Una locazione primaria del dato (sorgente).
  - Molte locazioni secondarie (destinazioni).
  - La modifica avviene solo nella locazione primaria.
- **Bidirezionale (two-way):**
  - Tutte le locazioni sono modificabili.
  - La modifica in una locazione si propaga a tutte le altre.

Altri Nomi

- State management
- Reactivity
- Observers/observable
- Watchable
- Publish/subscribe (pub/sub)
- Hooks

- Signals

Pattern di Programmazione

- **Model-View-Controller (MVC):**
  - Modello: natura concettuale del dato.
  - Vista: widget che mostra il dato.
  - Controller: allinea modello e vista, gestisce l'interazione.
- **Model-View-ViewModel (MVVM):**
  - Modello: come sopra.
  - Vista: come sopra.
  - ViewModel: collega elementi della vista alle strutture del modello.

Data Binding Fatto a Mano

- Esempi di implementazione di binding monodirezionale (MVC e MVVM).

Data Binding con Framework

- **Angular:**
  - Interpolazione: `{{ cliente.nome }}`
  - Binding di proprietà: `<img [src]="img.src">`
  - Binding di eventi: `<button (click)="onSave()">`
  - Binding bidirezionale: `<input [(ngModel)]="value">`
- **React:**
  - Interpolazione, proprietà, eventi.
  - Non ha binding bidirezionale nativo.
  - Hooks: `useState()`, `useContext()`, `useRef()`, `useEffect()`.
- **Vue:**
  - Data binding out of the box.
  - Reactivity API per la gestione dello stato condiviso tra componenti.

Browser: Mutation Observer

- Oggetto del browser per controllare le modifiche al DOM.
- Utile quando non si ha un controllo centralizzato delle modifiche.

Dipendenze Multiple

- Problema delle dipendenze incrociate tra dati del modello e oggetti di visualizzazione.
- Soluzione: albero di dipendenze esplicite (come nei fogli elettronici).
- Rilevamento di dipendenze circolari a compile time.

Signal

- Basato sul pattern observer.
- Valore osservabile con stato iniziale e funzione di aggiornamento.
- Emette un segnale quando il valore cambia, raggiungendo tutti i componenti dipendenti.
- Getter e setter per accedere e modificare il valore.
- Esempi in React e Angular.
- Efficienza nella propagazione delle modifiche, grazie all'albero delle dipendenze.