The Progressive JavaScript Framework

Comunicazione tra componenti

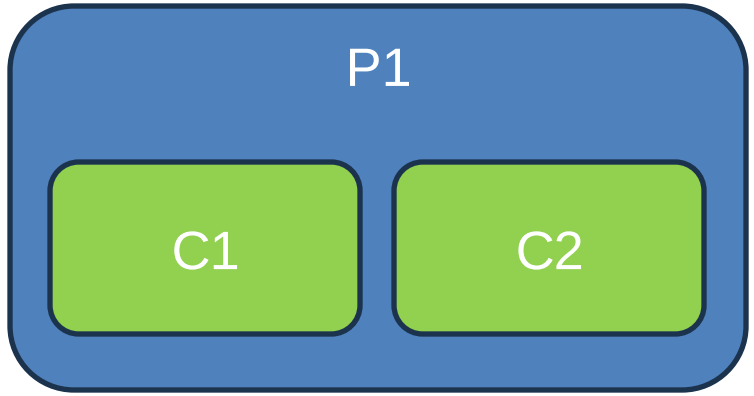
Nomenclatura

Dati più componenti disposti gerarchicamente, si identificano le seguenti relazioni con il rispettivo significato assunto:

- **Parent**: componente che contiene altri componenti, di cui ne è parent.
- **Child**: componente contenuto in un altro componente, di cui ne è child.
- **Sibling**: due componenti che condividono lo stesso parent.

Esempio

- P1 è **parent** di C1, C2
- C1, C2 sono **child(ren)** di P1
- C1 e C2 sono **siblings**



Comunicazione tra componenti

I componenti in Vue hanno la possibilità di comunicare tra loro mediante approcci differenti, ognuno con vincoli diversi.

La comunicazione è **unidirezionale** e può essere classificata in:

- Comunicazione **parent-child** (props, slots)
- Comunicazione **child-parent** (events)

Parent-child (props)

Un componente **parent** può passare delle proprietà (**props**) ad un componente **child**.

Questo meccanismo garantisce il riutilizzo del componente, rendendolo parametrizzabile.

Il componente **child** deve rendere esplicito la presenza delle proprietà (come se esponesse un'**interfaccia**)

```
<div id="app">
    Hello
    <strong-text message="my friend"></strong-text>
    and welcome to
    <strong-text message="this example"></strong-text>
</div>

<script>
    const StrongText = Vue.defineComponent({
        template: '<strong>{{ message }}</strong>',
        props: { message: String }
    });

    Vue.createApp({
        components: { StrongText }
    }).mount('#app');
</script>
```

examples/example-01/simple-props.html

Parent-child (props)

Un componente **parent** può effettuare *bind* di una proprietà verso il **child**.

La proprietà del child diventerà reattiva, garantendo modifiche dello stato (e della visualizzazione).

Un child **non** può modificare il valore delle props (sono in **sola lettura**).

```
<div id="app">
  You click the button:
  <strong-text v-bind:message="count"></strong-text>
  <br>
  <button v-on:click="increment">Click me!</button>
</div>

<script>
  const StrongText = Vue.defineComponent({
      template: '<strong>{{ message }}</strong>',
      props: { message: Number }
  });

  Vue.createApp({
      data() { return { count: 0 } },
      methods: {
          increment() { this.count++; }
      },
      components: { StrongText }
  }).mount('#app');
</script>
```

examples/example-01/bind-props.html

Parent-child (props)

Per la comunicazione **parent-child**, mediante **binding** di proprietà si ritrovano le seguenti fasi:

Child

1. Definizione delle proprietà (interfaccia)
2. Utilizzo delle proprietà in sola lettura

Parent

3. Definizione stato reattivo
4. Bind dello stato alla proprietà del child

```
<div id="app">
  You click the button:  ④
  <strong-text v-bind:message="count"></strong-text>
  <br>
  <button v-on:click="increment">Click me!</button>
</div>

<script>
  const StrongText = Vue.defineComponent({
      template: '<strong>{{ message }}</strong>',
      props: { message: Number }
  });

  Vue.createApp({
      data() { return { count: 0 } },
      methods: {
          increment() { this.count++; }
      },
      components: { StrongText }
  }).mount('#app');
</script>
```

examples/example-01/bind-props.html

Parent-child (bad practice)

Un componente child può modificare il contenuto di una **proprietà** passata per **referenza** (Object, Array). Ciò è reso possibile dal linguaggio, ma è considerata una **bad practice**.

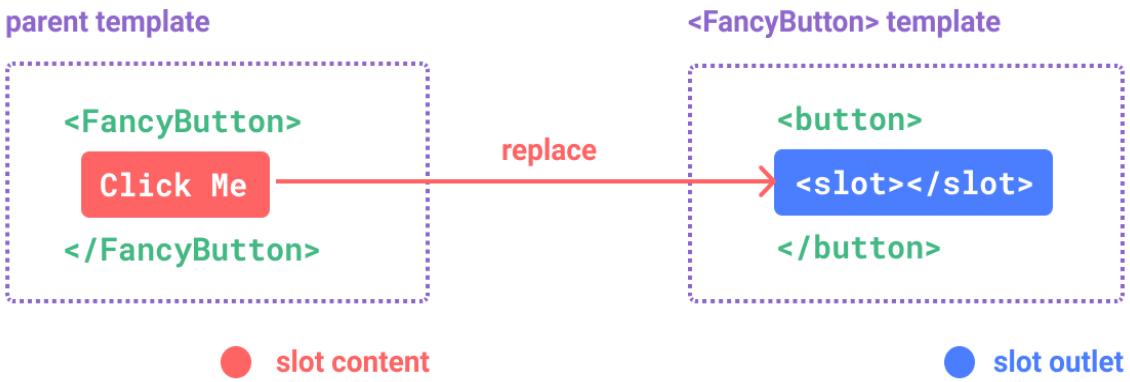
```
<script>
  const StrongText = Vue.defineComponent({
    template: `...`,
    props: {
      message: Number,
      innerProp: Object
    },
    methods: {
      tryToModifyPrimitive() {
          this.message++; // [warn]Props are readonly
      },
      tryToModifyReference() {
          this.innerProp.name = 'Hello from Inner';
      }
    }
  });
  // ...
</script>
```

examples/example-01/bad-practices.html

Parent-child (slots)

Un altro metodo per realizzare una comunicazione **parent-child** è l'utilizzo di **slot**.

Ciò permette al **parent** di specificare il **contenuto** del tag **child**.




```html
<div id="app">
    You click the button:
    <strong-text>{{ count }} times</strong-text>
    <br>
    <button v-on:click="increment">Click me!</button>
</div>

<script>
    const StrongText = Vue.defineComponent({
        template: '<strong><slot></slot></strong>'
    });

    Vue.createApp({
        data() { return { count: 0 } },
        methods: {
            increment() { this.count++; }
        },
        components: { StrongText }
    }).mount('#app');
</script>
```

examples/example-01/slot.html

Child-parent

Un componente **child** può inviare degli **eventi** al proprio componente **parent**.

Il componente parent si sottoscrive all'evento e definisce il comportamento da eseguire al verificarsi di esso

```
<div id="app">
  You click the button: {{ count }} times.<br>
  <strong-button v-on:clicked-event="handleClick">
  </strong-button>
</div>
```

```
<script>
  const StrongButton = Vue.defineComponent({
    template: `
    <button v-on:click="clicked">
      <strong>Click me!</strong>
    </button>`,
    methods: {
      clicked() {
        this.$emit('clicked-event');
      }
    }
  });

  Vue.createApp({
    data() { return { count: 0 } },
    components: { StrongButton },
    methods: {
      handleClick() { this.count++; }
    },
  }).mount('#app');
</script>
```

examples/example-02/event.html

Child-parent

Fasi comunicazione **child-parent**

Child

- Emette un evento mediante metodo `$emit` con un determinato nome

Parent

- Si sottoscrive all'evento generato dal **child**

```
<div id="app">
  You click the button: {{ count }} times.<br>
  <strong-button v-on:clicked-event="handleClick">
  </strong-button>
</div>
```

```
<script>
  const StrongButton = Vue.defineComponent({
    template: `
    <button v-on:click="clicked">
      <strong>Click me!</strong>
    </button>`,
    methods: {
      clicked() {
        this.$emit('clicked-event');
      }
    }
  });

  Vue.createApp({
    data() { return { count: 0 } },
    components: { StrongButton },
    methods: {
      handleClick() { this.count++; }
    },
  }).mount('#app');
</script>
```

examples/example-02/event.html

Child-parent

Gli **eventi** possono avere anche **argomenti**.

Nel metodo

`$emit(name, arg1, arg2, …, argN)`

il primo argomento è il **nome** dell'evento. Gli altri sono passati alla funzione di handler come **parametri**.

```
<div id="app">
  You click the button: {{ count }} times.<br>
  <strong-button v-on:clicked-event="handleClick">
  </strong-button>
</div>
```

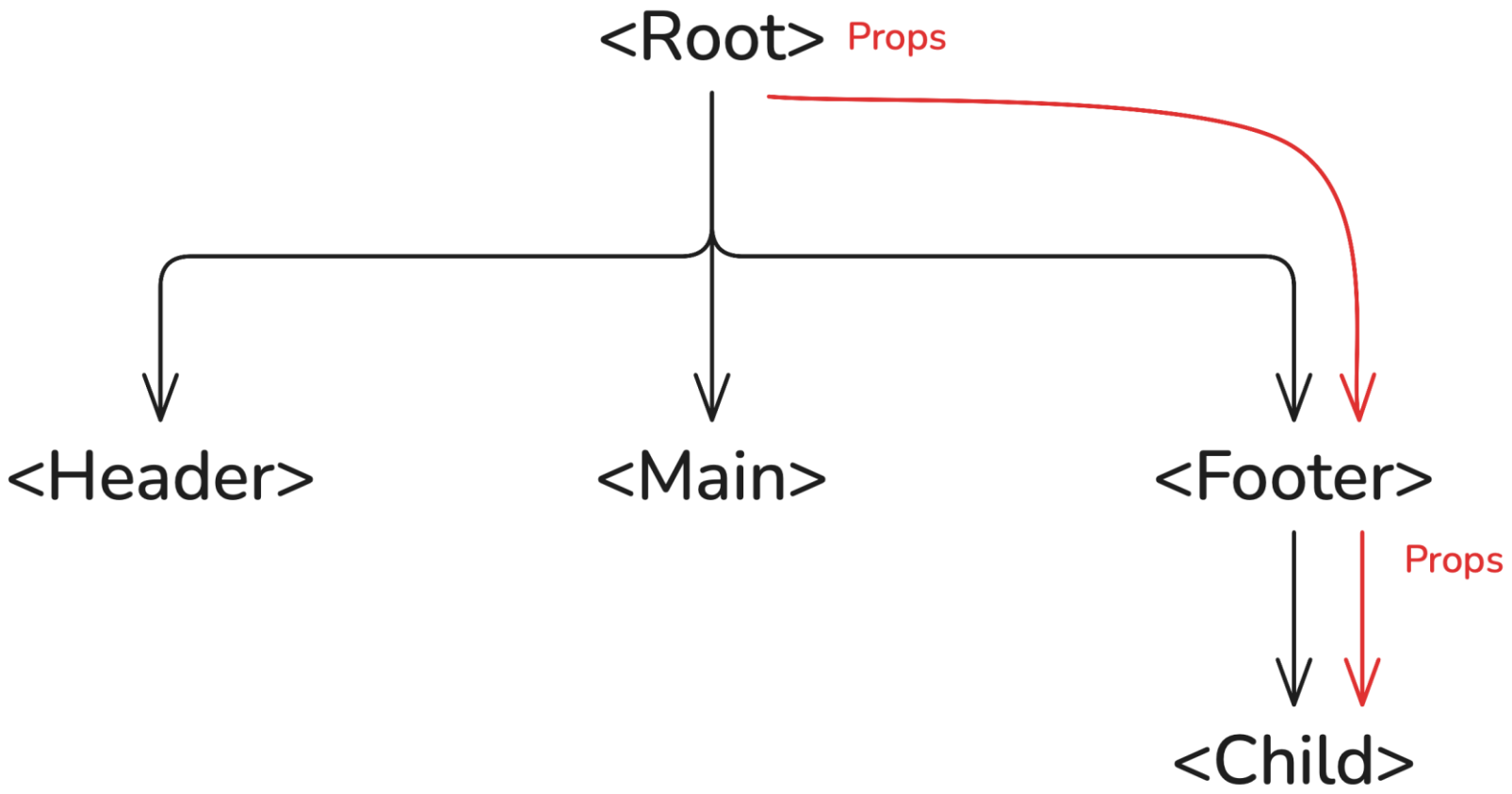
```
<script>
  const StrongButtonWithRandom = Vue.defineComponent({
      template: `...`,
      methods: {
          clicked() {
              this.$emit('clicked-event', Math.random());
          }
      },
  });

  Vue.createApp({
      data() { return { count: 0 } },
      components: { StrongButtonWithRandom },
      methods: {
          handleClick(randomNumber) {
              this.count = randomNumber;
          }
      },
  }).mount('#app');
</script>
```

examples/example-02/parameters.html

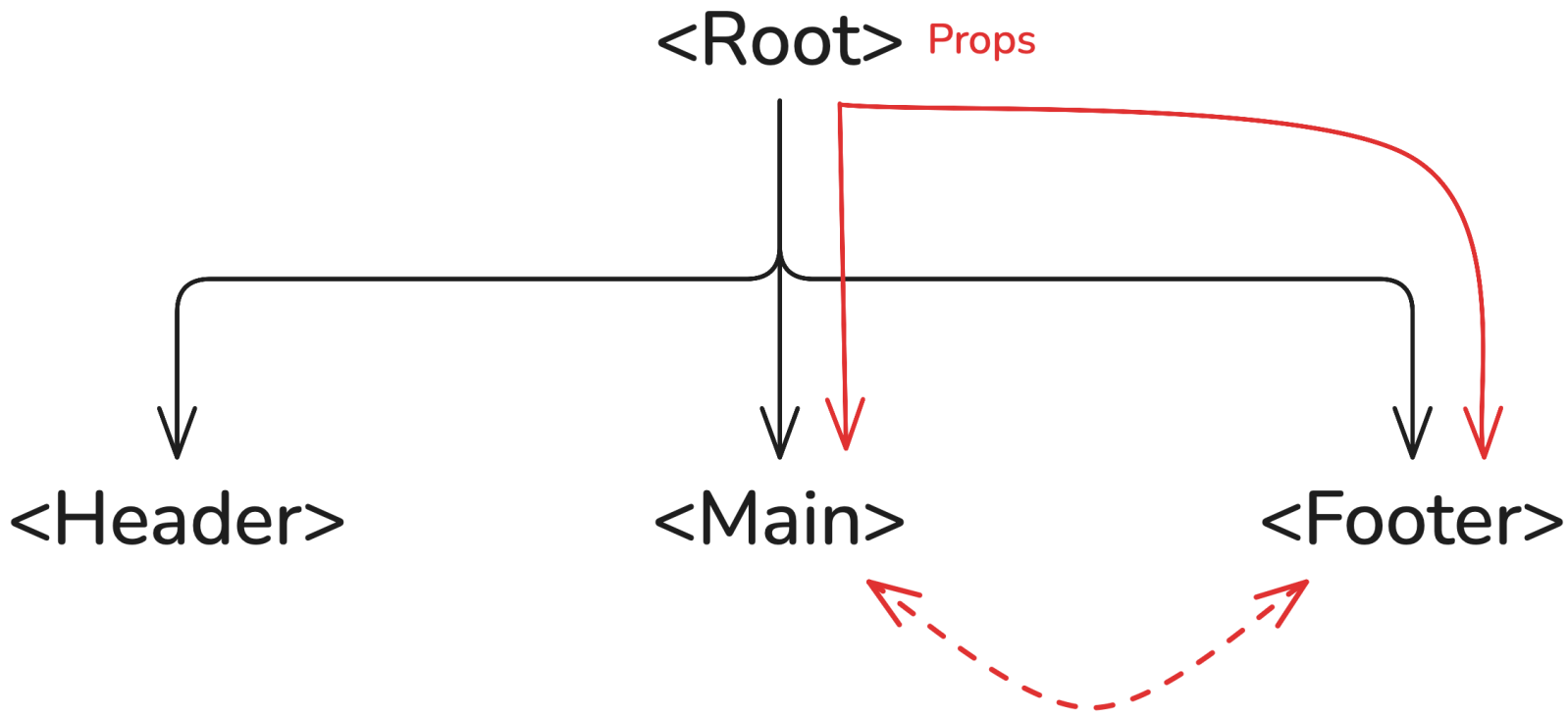
Scenari tipici

Comunicazione con i discendenti



Scenari tipici

Comunicazione tra siblings



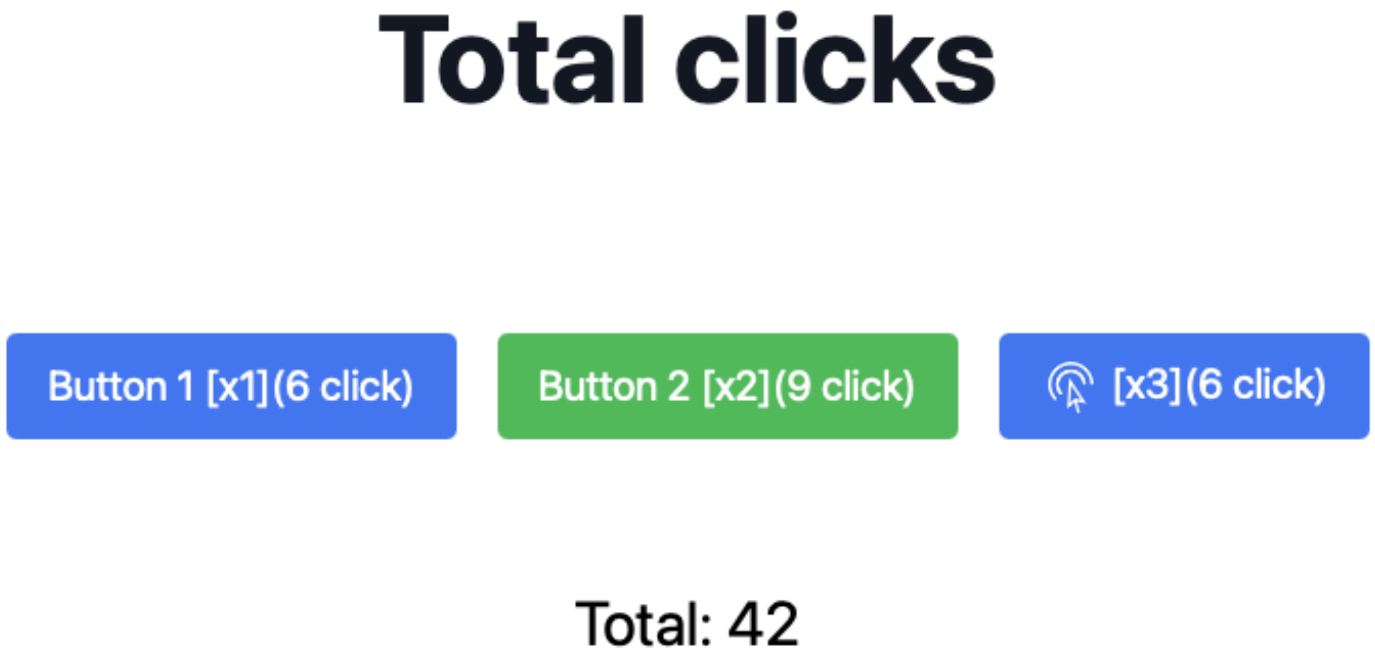
Esercizio 1

Realizzare una schermata come in figura, utilizzando le funzionalità di Vue viste finora, con la seguente logica.

Sono presenti **3 bottoni** che mostrano il numero di click ricevuti. Ogni bottone ha assegnato un **moltiplicatore**, ovvero il contributo che porta ogni suo click al totale.
E.g. cliccando un bottone con moltiplicatore 3 ([x3]), il totale verrà incrementato di 3.
In verde viene mostrato l'ultimo bottone cliccato.

Nella cartella `exercises/exercise-01` è presente uno scheletro iniziale

Esercizio 1


Total clicks
Button 1 [x1](6 click)
Button 2 [x2](9 click)
[x3](6 click)
Total: 42

Single-File Component (SFC)

Un **Single-File Component** (SFC, estensione .vue), è un formato di file che utilizza una sintassi simile all'HTML per descrivere un componente Vue.
Mirano ad incapsulare in un unico file i concetti di **markup**, **logica** e **stile** di un singolo componente.

È possibile utilizzare i SFC mediante **Option API** e **Composition API**.

Il contenuto di questi file viene **compilato** dal compilatore di Vue, che produce in output file standard di JavaScript e CSS.

Single-File Component (SFC)

Ogni file è composto da principalmente da tre tipi di blocchi:

- `<template>`: definisce il **markup** del componente (HTML, direttive)
- `<script>`: definisce la **logica** del componente (JavaScript, TypeScript)
- `<style>`: definisce lo **stile** del componente (CSS, SCSS)

```vue
<script>
export default {
  data() {
    return {
      greeting: 'Hello World!'
    }
  }
}
</script>

<template>
  <p class="greeting">{{ greeting }}</p>
</template>

<style>
.greeting {
  color: red;
  font-weight: bold;
}
</style>
```

Option API

Single-File Component (init)

Per creare un nuovo progetto con la possibilità di utilizzare i SFC è possibile procedere mediante il seguente comando:

```
npm create vue@latest
```

Dopo una fase di configurazione, verrà generato un nuovo progetto con esempi di utilizzo.

Suggerimento: installare l'estensione di Vue per VSCode (**vue.volar**).

Single-File Component (vite)

Vue adotta **Vite**, un build tool per lo sviluppo e il bundling di progetti in ambito web. I principali comandi utili sono:

- `npm run dev` (o `vite`): avvia un web server per lo sviluppo che modificherà la pagina automaticamente ai cambiamenti del codice
- `npm run build` (o `vite build`): compila il progetto per la produzione con output nella cartella `./dist`
- `npm run preview` (o `vite preview`): avvia un web server che mostra il progetto compilato

Esercizio 2

Implementare l’esercizio 1 attraverso l’utilizzo delle **Option API** e dei **Single-File Components**.

Esercizio 2

- Posizionarsi con il terminale nella cartella `exercises/exercise-02` dove è presente uno scheletro per l'esercizio
- Installare le dipendenze del progetto: `npm install`
- Eseguire il progetto in modalità sviluppo: `npm run dev`

Esercizio 3

Implementare **uno o più componenti** Vue, utilizzando le Option API e i Single-File Components, realizzando una **tabella** a partire da dataset a scelta.

La tabella dovrà essere realizzata in due versioni:

- Tabella **minimale**
- Tabella **accessibile**

Utilizzare le funzionalità di Vue il più possibile!

È caldamente consigliata la **consegna** su virtuale!

Esercizio 3 (client)

- Posizionarsi con il terminale nella cartella `exercises/exercise-03` dove è presente lo scheletro dell'esercizio
- Installare le dipendenze del progetto: `npm install`
- Eseguire il progetto in modalità sviluppo: `npm run dev`

Esercizio 3 (server)

- Scaricare il dataset scelto
- Mettere il dataset nella cartella `exercises/exercise-03/dataset`
- Eseguire un web server per servire il dataset

```
npm install --global serve
npx serve --cors dataset
```

Esercizio 3 (istruzioni)

1. Scelta del dataset
   - Il dataset deve essere diverso per ogni studente
   - Cardinalità minima del dataset: 50x5 (50 record e 5 features)
   - Esempi di dataset: canzoni, film, misurazioni ambientali, etc.
   - Alcune fonti di dataset:
     - https://github.com/jdorfman/awesome-json-datasets
     - https://www.kaggle.com/datasets/
     - https://www.openml.org/search?type=data

Esercizio 3 (istruzioni)

2. Registrare il dataset scelto su virtuale
   - Sezione "Consegna Laboratorio" > "Dataset utilizzati"
   - Compilare i campi richiesti
   - Controllare che altri non abbiano scelto lo stesso dataset!

Esercizio 3 (istruzioni)

3. Consegnare la soluzione su virtuale mediante l'apposita sezione
   - Consegnare l'intero progetto (`exercises/exercise-03`) in formato zip
   - Includere anche il **dataset**
     - Ridurre la cardinalità qualora superi lo spazio disponibile (20 MB)
   - Rimuovere eventuali file generati
     - `node_modules`
     - `dist`