



UNIVERSITÀ DEGLI STUDI DI CAGLIARI

CORSO DI LAUREA IN INFORMATICA

Silvia Maria Massa – silviam.massa@unica.it

Fondamenti di Programmazione Web

Client (3), Gestione dello stato e Autenticazione



Vue.js Computed

- È frequente dover visualizzare nei template dati che possono essere dedotti da altri o che richiedono una particolare formattazione.

```
data() {  
  return {  
    ultimaModifica: new Date()  
  };  
},
```

script

Wed May 14 2025 10:23:45 GMT+0200 (Central European Summer Time)

```
computed: {  
  ultimaModificaFormattata() {  
    return this.ultimaModifica.toLocaleDateString('it-IT', {  
      weekday: 'long',  
      year: 'numeric',  
      month: 'long',  
      day: 'numeric'  
    });  
  }  
}
```

martedì 14 maggio 2025

```
<p>Ultima modifica: {{ ultimaModificaFormattata }}</p>
```

template

Vue.js Computed

- Anche se è possibile inserire espressioni JavaScript direttamente all'interno delle direttive del template, è buona pratica evitare codice complesso o duplicato, delegando questi calcoli a proprietà calcolate (**computed**).
- Una proprietà calcolata è una proprietà che può essere derivata in modo sincrono da altre proprietà disponibili (props, data o altre proprietà computed).
- Vue è in grado di determinare automaticamente da quali dati dipende una proprietà calcolata, aggiornandone il valore solo quando necessario, grazie al suo sistema di reattività.

Vue.js Watchers

- Vue consente anche di definire osservatori (**watchers**) che eseguono automaticamente del codice ogni volta che cambia il valore di una proprietà reattiva, come una prop, un dato in data, o una proprietà computed.
- I watcher si dichiarano all'interno della sezione **watch** del componente.
- Ogni watcher è una funzione associata a una specifica proprietà da osservare.
- Quando il valore di quella proprietà cambia, il watcher viene automaticamente chiamato, ricevendo come argomenti sia il nuovo valore sia il valore precedente.

Vue.js Computed and Watchers

```
<script>
export default {
  data() {
    return {
      maggiorenni: 10,
      minorenni: 25,
      lastUpdate: null
    };
  },
  computed: {
    total() { return this.maggiorenni + this.minorenni; }
  },
  watch: {
    maggiorenni() { this.lastUpdate = new Date(); },
    minorenni() { this.lastUpdate = new Date(); }
  }
};
</script>
```

computed: Utile quando serve ottenere un risultato da visualizzare o usare direttamente nel template. Restituisce un valore derivato da altre proprietà reattive.

watch: Utile quando si vuole reagire a cambiamenti dei dati in modo personalizzato. Non restituisce un valore, ma esegue codice.

```
<template>
  <div>
    <p>Numero di iscritti al corso:</p>
    <span>{{ total }}</span>
    <p>Ultimo aggiornamento:</p>
    <span>{{ lastUpdate }}</span>
  </div>
</template>
```

Vue.js Ciclo di vita del componente

- Durante il ciclo di vita di un componente vengono eseguite delle funzioni di callback chiamate hook del ciclo di vita.
- Queste offrono l'opportunità di eseguire codice in momenti specifici della vita del componente, ad esempio all'inizializzazione, al montaggio nel DOM, durante gli aggiornamenti e alla distruzione.
- Le fasi principali sono:
 - **beforeCreate** e **created** chiamate prima e dopo l'inizializzazione del componente
 - **beforeMount** e **mounted** chiamate prima e dopo l'aggiunta del componente al DOM
 - **beforeUpdate** e **updated** chiamate ogni volta che c'è una modifica nei dati del nostro componente, ma prima che l'aggiornamento venga reso sullo schermo e dopo che il nostro componente ha aggiornato il suo albero DOM
 - **beforeDestroy** (Vue 2)/ **beforeUnmount** (Vue 3) e **destroyed** (Vue 2)/ **unmounted** (Vue 3) chiamate prima e dopo che un componente viene rimosso dal DOM

Vue.js Ciclo di vita del componente

- Queste callback possono essere usate per definire un comportamento specifico per il componente in questi momenti precisi.
- Nello specifico:
 - created – usato per inizializzare lo stato o eseguire logica prima che il componente sia montato.

```
created() {  
    console.log('Componente creato');  
    this.messaggio = 'Benvenuto!'  
}
```
 - mounted – usato quando è necessario accedere o modificare il DOM.
 - destroyed (Vue 2) / unmounted (Vue 3) – usato per rimuovere timer (es. `setInterval()` e `setTimeout()`) o altre risorse quando il componente viene rimosso per evitare rallentamenti, bug e perdite di memoria nel tempo.
- Altri callback sono riservati a casi d'uso più specifici.



Comunicazione tra componenti

Comunicazione da padre a figlio (props)

- Come qualsiasi altro elemento HTML, i componenti di Vue possono ricevere argomenti, chiamati **props** o proprietà.
 - elemento HTML ``
 - componente di Vue `<ProfiloUtente :nome="utente.nome" :eta="utente.eta" />`
- Le **props** sono usate per trasmettere informazioni da un componente padre a un componente figlio.
- È necessario dichiarare l'elenco delle proprietà accettate nell'opzione **props** del componente figlio.

```
export default {  
  props: ['nome', 'eta'],  
  ....  
}
```

componente figlio ProfiloUtente.vue

- Le proprietà ricevute possono essere utilizzate nel **template** o in **methods** proprio come le proprietà dichiarate in data.
- La differenza è che eviteremo di riassegnare o mutare i props: poiché questi valori provengono dal componente genitore.
 - Dobbiamo piuttosto comunicare con questo genitore (comunicazione ascendente), in modo che apporti lui stesso la modifica.
 - Il valore modificato sarà poi riportato automaticamente ai componenti figli.



Comunicazione da padre a figlio (props)

```
<!-- BlogPost.vue -->
<template>
  <article>
    <h3>{{ title }}</h3>
    <p>{{ content }}</p>
  </article>
</template>

<script>
export default {
  props: ["title", "content"]
};
</script>
```

COMPONENTE FIGLIO

```
<template>
<!-- in parent component template -->
<blog-post :title="article.title" :content="article.content" />
<!-- equivalent shorthand syntax -->
<blog-post v-bind="article" />
</template>

<script>
import blog-post from "./BlogPost.vue ";

export default {
  data() {
    return {
      article: {title: 'La vita nello spazio' , content: 'Solitaria.'},
    };
  },
  components: {blog-post},
};
</script>
```

COMPONENTE PADRE

Comunicazione da padre a figlio (props)

- Opzionalmente, è possibile specificare il tipo di props o fornire opzioni di validazione.
- Vue rifiuterà i valori non validi per props con messaggi di errore espliciti, il che è utile quando si utilizzano componenti di terze parti.

```
<script>
  export default {
    props: {
      //propA accetta un valore di tipo Number
      propA: Number,
      //propB accetta String o Number
      propB: [String, Number],
      //propC ha tipo String e un valore di default
      propC: {type: String, default: "test"},
      //propD è obbligatoria e deve iniziare con "_"
      propD: {required: true, validator: value => value.startsWith("_")}
    }
  };
</script>
```

Comunicazione da figlio a padre (eventi)

- I componenti figli comunicano con i loro genitori utilizzando gli **eventi**: emettono eventi che si propagano da genitore a genitore, allo stesso modo degli eventi DOM come il clic del mouse.
- Un buon componente è agnostico rispetto al suo ambiente, non conosce i suoi genitori e non sa se gli eventi che emette saranno mai intercettati (o "ascoltati").
- Per emettere un evento, si usa il metodo **\$emit**, disponibile in tutti i componenti di Vue. Esso prende come parametro il nome dell'evento e, facoltativamente, un valore (payload) da trasmettere.
- Se è necessario passare più valori, si incapsulano in un oggetto.
- L'elenco degli eventi inviati da un componente dovrebbe essere descritto nell'opzione **emits** del componente, non obbligatoria ma utile a fini di documentazione.
- Per ascoltare un evento emesso da un componente figlio, si usa la stessa direttiva **v-on** degli eventi DOM, o **@yourEvent**.

Comunicazione da figlio a padre (eventi)

```
<template>                                     COMPONENTE FIGLIO
  <article>
    <h3>My article</h3>
    <p>Lorem ipsum...</p>
    <textarea v-model="comment" />
    <button @click="sendComment">Comment</button>
  </article>
</template>

<script>
export default {
  data() {
    return { comment: "" };
  },
  emits: ['comment'],
  methods: {
    sendComment() { this.$emit("comment", this.comment); }
  }
};
</script>
```

Comunicazione da figlio a padre (eventi)

```
<template>
```

COMPONENTE PADRE

```
<div>
```

```
<h2>Commenti ricevuti:</h2>
```

```
<ul>
```

```
<li v-for="(commento, index) in commenti" :key="index">{{ commento }}</li>
```

```
</ul>
```

```
<blog-post @comment="onNewComment" />
```

```
</div>
```

```
</template>
```

```
<script>
```

```
import BlogPost from './BlogPost.vue'; // importa il componente figlio
```

```
export default {
```

```
  components: { BlogPost },
```

```
  data() { return
```

```
    {commenti: [] // array che raccoglie i commenti ricevuti};
```

```
  },
```

```
  methods: { onNewComment(testo) { this.commenti.push(testo); } }
```

```
};
```

```
</script>
```



Complete Options API of Vue Components

<script>

```
export default {
```

```
  name: "MyComponent", // useful for debugging purposes
```

```
  components: {}, // declared child components
```

```
  props: {}, // properties passed from parent
```

```
  data() {}, // component internal state variables
```

```
  computed: {}, // computed properties
```

```
  watch: {}, // observed properties
```

```
  methods: {}, // component own methods
```

```
  emits: [], // events emitted by this component
```

```
  .....
```

```
};
```

</script>



Complete Options API of Vue Components

```
<script>
  export default {
    .....
    // component lifecycle hooks
    beforeCreate() {},
    created() {},
    beforeMount() {},
    mounted() {},
    beforeUpdate() {},
    updated() {},
    beforeDestroy() {}, // beforeUnmount with Vue 3
    destroyed() {}, // unmounted with Vue 3
    .....
  };
</script>
```


Composition API: Struttura Base di un Componente

```
<script setup>
```

```
import { ref, reactive, computed, watch, onMounted, onBeforeMount, onUpdated,
onBeforeUpdate, onUnmounted, onBeforeUnmount, defineProps, defineEmits }
from 'vue';
```

```
// Props
```

```
const props = defineProps({ // es. title: String });
```

```
// Emits
```

```
const emit = defineEmits(['custom-event']);
```

```
// Reactive state
```

```
const count = ref(0);
```

```
const state = reactive({ name: 'Vue' });
```

```
// Computed
```

```
const double = computed(() => count.value * 2);
```

```
.....
```

```
</script>
```

Composition API: Struttura Base di un Componente

<script setup>

.....

// Watch

```
watch(() => count.value, (newVal, oldVal) => {  
  console.log(`Count changed from ${oldVal} to ${newVal}`);  
});
```

// Methods (normali funzioni nel setup)

```
function increment() { count.value++; }
```

// Lifecycle hooks

```
onBeforeMount(() => { console.log('beforeMount'); });  
onMounted(() => { console.log('mounted'); });  
onBeforeUpdate(() => { console.log('beforeUpdate'); });  
onUpdated(() => { console.log('updated'); });  
onBeforeUnmount(() => { console.log('beforeUnmount'); });  
onUnmounted(() => { console.log('unmounted'); });
```

.....

</script>

Routing

- Le applicazioni Vue sono prevalentemente Single Page Application (SPA).
- In un'architettura SPA, il server serve sempre una singola pagina HTML e la navigazione tra le pagine/sezioni dell'applicazione è gestita dal client tramite JavaScript.
- Questo approccio consente transizioni più fluide tra le pagine e riduce il numero di chiamate al server necessarie per navigare tra le pagine perché non si devono aggiornare tutte le parti della pagine.
- Il routing in una SPA collega l'URL del browser al contenuto visualizzato dall'utente. Mentre l'utente naviga nell'applicazione, l'URL si aggiorna senza richiedere il ricaricamento della pagina dal server.
- Vue offre una libreria specifica per il routing chiamata **vue-router**

Routing - installazione

- Se non è stato installato durante la configurazione iniziale del progetto, è possibile installare vue-router con npm.

npm install vue-router

- Si crea quindi una cartella **src/router** e un file **index.js** per contenere la configurazione del router.
- Il file main.js dovrà essere modificato per dichiarare questo nuovo router nell'applicazione:

```
import { createApp } from 'vue'  
import { createPinia } from 'pinia'  
import App from './App.vue'  
import router from "./router";
```

```
const pinia = createPinia();
```

```
createApp(App)  
  .use(pinia)  
  .use(router)  
  .mount("#app")
```

Routing - configurazione

- Il router viene creato passando come parametri a **createRouter()** un elenco di route.
- Ogni route associa un URL a una determinata view.
- Ogni volta che la pagina viene caricata o l'URL cambia, Vue router cerca una route che corrisponda al nuovo indirizzo e mostra il componente collegato a quella route.

```
/** src/router/index.js */  
import { createRouter, createWebHistory } from 'vue-router'  
import HelloWorld from "@components/HelloWorld.vue";  
  
const router = createRouter({  
  history: createWebHistory(), /*Indica come Vue Router gestisce gli URL.  
                                createWebHistory() è il metodo consigliato per le SPA moderne. */  
  routes: [  
    {  
      path: "/hello/:name", //URL associato alla route  
      name: "hello", //nome identificativo alla route  
      component: HelloWorld //componente collegato  
    }  
  ]  
})  
export default router;
```

Routing - configurazione

- Una volta completata la ricerca della route, un componente viene associato all'URL corrente.
- Questo componente viene quindi iniettato al posto dell'elemento **<router-view />**.
- Questo elemento è solitamente collocato nel componente principale App.vue.
- Gli elementi intorno a **<router-view />** formano il layout che struttura l'applicazione: un header, una navbar, un footer, ecc. (che solitamente sono comuni a tutte le pagine del vostro sito).

```
<template>
  <div class="app">
    <header><h1>My website</h1></header>
    <router-view />
    <footer>Made with Vue</footer>
  </div>
</template>
```

Navigazione e router-link

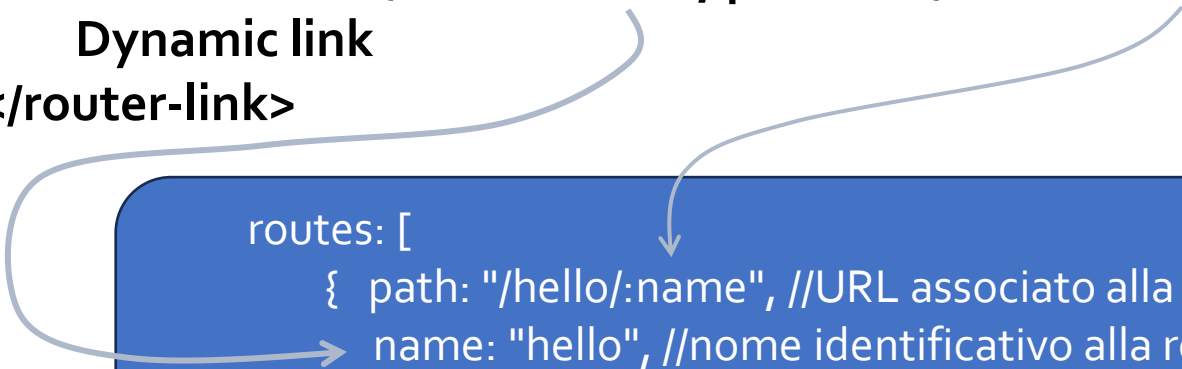
- Vue-router include un componente **<router-link>** dichiarato a livello globale, che può sostituire i tag **<a>** per qualsiasi navigazione interna effettuata tramite questo router.
- **<router-link>** permette a Vue router di cambiare l'URL senza ricaricare la pagina.
- Inoltre con i router-link i path potranno essere statici o generati dinamicamente dai nomi delle route e dagli elenchi di parametri:

<router-link to="/home">Homepage</router-link>

<router-link :to="{ name: 'hello', params: { name: 'John' } }">

Dynamic link

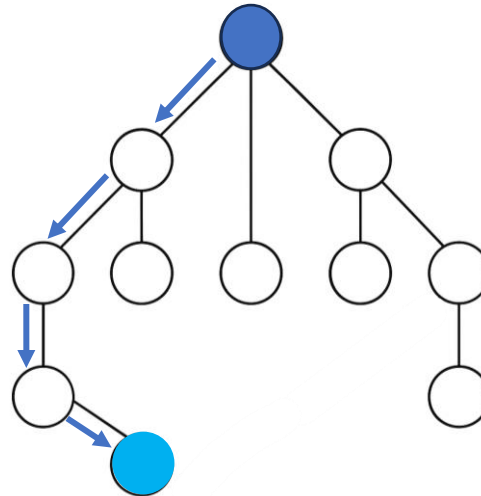
</router-link>



```
routes: [  
  { path: "/hello/:name", //URL associato alla route  
    name: "hello", //nome identificativo alla route  
    component: HelloWorld //componente collegato }  
]
```

Gestione dello Stato

- Quando le applicazioni crescono e diventano più complesse, i componenti che sono lontani l'uno dall'altro nell'albero dei componenti possono dover manipolare gli stessi dati.



- I metodi che abbiamo visto per far comunicare padre e figlio non sono adatti.
- Sono state proposte diverse soluzioni per la **gestione dello stato** più o meno complesse. Noi vedremo **Pinia**.

Gestione dello Stato – perché?

- Per evitare che un dato per essere manipolato da più componenti lontani debba essere propagato da un componente genitore ai componenti nipoti e pronipoti.
- Per poter condividere le informazioni tra diversi alberi di componenti diversi.
- Per delegare la gestione dei dati a un servizio raggiungibile da tutti i componenti.
- Per essere in grado di rendere persistenti i dati automaticamente (ad esempio, in *localStorage*).
- Per registrare gli stati dell'applicazione o tornare a uno stato precedente con una funzione di annullamento (Cancel).
- Per rendere il debug più semplice.

Pinia

- La comunità di Vue ha proposto prima **Vuex** come soluzione ufficiale di gestione degli stati per Vue.
- Dopo il rilascio di Vue 3 e l'introduzione dell'API Composition, una nuova libreria chiamata **Pinia** ha sostituito Vuex.
- Attualmente Pinia è la soluzione ufficiale per la gestione degli stati fornita dal team di Vue.
- Pinia non troverà necessariamente posto in tutti i progetti Vue, ma è uno strumento molto efficiente nelle applicazioni di grandi dimensioni che gestiscono molti dati.

Pinia

- Uno store viene definito tramite la funzione **defineStore()**
- Per convenzione, il nome della funzione che richiama lo store segue il formato: **'use + nome dello store con lettera maiuscola + Store'**
- Il primo parametro passato è un identificatore univoco per lo store all'interno dell'intera applicazione
- Il secondo parametro può essere una funzione di setup (simile Composition API), o un Options Object.(simile Options API)
- L'**Options Object** è molto simile all'Options API di Vue:
 - la proprietà **state** funziona come **data** dei componenti,
 - la proprietà **getters** funziona come **computed** dei componenti,
 - la proprietà **actions** funziona come i **methods** dei componenti.
- Lo **state** mutato aggiorna reattivamente tutte le viste che lo utilizzano, indipendentemente dalla loro profondità nell'albero dei componenti.



Pinia – es. definizione store (Options Store)

```
import { defineStore } from 'pinia' counter.js

export const useCounterStore = defineStore('counter', {
  state: () => ({
    count: 0
  }),
  getters: {
    doubleCount: (state) => state.count * 2,
    doubleCountPlusOne() { return this.doubleCount + 1 },
  },
  actions: {
    increment() {this.count++ },
  },
})
```



Pinia – es. definizione store (Setup Store)

```
import { defineStore } from 'pinia'
import { ref, computed } from 'vue'
```

counter.js

```
export const useCounterStore = defineStore('counter', () => {
  const count = ref(0) //State
```

```
  //Getters
```

```
  const doubleCount = computed(() => count.value * 2)
```

```
  const doubleCountPlusOne = computed(() => doubleCount.value + 1)
```

```
  //Actions
```

```
  function increment() {
```

```
    count.value++
```

```
  }
```

```
  return { count, doubleCount, doubleCountPlusOne, increment }
```

```
})
```

Pinia – es. utilizzo store (Options API)

```
<script>
import { useCounterStore } from '@stores/counter'

export default {
  data() {
    return {
      counter: useCounterStore()
    }
  },
  methods: {
    incrementAndPrint() {
      this.counter.increment()
      console.log('New Count:', this.counter.count)
    },
  },
}
</script>
```

```
<template>
  <div>
    <p>Double count is {{ counter.doubleCount }}</p>
    <p>Double count + 1 is {{ counter.doubleCountPlusOne }}</p>
    <button @click="incrementAndPrint">Increment</button>
  </div>
</template>
```



Pinia – es. utilizzo store (Composition API)

```
<script setup>
```

```
import { useCounterStore } from '@stores/counter'
```

```
// per accedere alla variabile `store` in qualsiasi punto del componente  
const counter = useCounterStore()
```

```
function incrementAndPrint() {  
  counter.increment();  
  console.log(`New Count: `, counter.count);  
}
```

```
</script>
```

```
<template>  
  <div>  
    <p>Double count is {{ counter.doubleCount }}</p>  
    <p>Double count + 1 is {{ counter.doubleCountPlusOne }}</p>  
    <button @click="incrementAndPrint">Increment</button>  
  </div>  
</template>
```

Pinia, LocalStorage e SessionStorage

- E se volessimo conservare alcune informazioni anche dopo che l'utente ha ricaricato la pagina o ha chiuso e riaperto il browser?
- Gli oggetti **localStorage** e **sessionStorage** permettono di salvare le coppie key/value nel browser.
- localStorage - memorizza i dati senza data di scadenza
- sessionStorage - memorizza i dati per una sessione (i dati vengono persi quando la scheda del browser viene chiusa)
- I metodi disponibili per questi oggetti sono:
 - `setItem(key, value)`: memorizza la coppia key/value.
 - `getItem(key)`: lettura del valore dalla key.
 - `removeItem(key)`: rimuove la key, ed il relativo value.
 - `clear()`: rimuove tutti gli elementi.
- Es. `localStorage.setItem('nomeUtente', nome);`

Pinia, LocalStorage e SessionStorage

- Se si desidera una soluzione pronta all'uso, il plugin **pinia-plugin-persistedstate** è stato creato appositamente per gestire la persistenza degli stati con Pinia.

Vantaggi uso di *pinia-plugin-persistedstate* :

- Persistenza automatica dello stato
 - Lo stato definito nello store viene salvato in automatico ogni volta che cambia.
 - Nessun bisogno di scrivere codice extra per salvare o caricare dati.
- Codice più pulito e mantenibile
 - Tutta la logica di persistenza è gestita automaticamente dal plugin.
 - Il codice degli store resta semplice e focalizzato sullo stato e le azioni.
- Reattività garantita
 - Tutti i dati persistiti rimangono reattivi e integrati nel flusso dell'app Vue.
- Configurazione flessibile
 - Puoi scegliere quali proprietà dello store persistere.
 - Puoi decidere se usare localStorage o sessionStorage.

pinia-plugin-persistedstate

- Si installa il plugin (se non è stato installato durante la configurazione iniziale del progetto) ***npm install pinia-plugin-persistedstate***
- In src/main.js aggiungere le seguenti configurazioni

```
import { createApp } from "vue"
import { createPinia } from "pinia"
import piniaPluginPersistedstate from "pinia-plugin-persistedstate" //Add your plugin import

import App from "./App.vue";

const app = createApp(App);
const pinia = createPinia(); // initialize Pinia

//Use the plugin
pinia.use(piniaPluginPersistedstate);
app.use(pinia);
app.mount("#app");
```

pinia-plugin-persistedstate

- Nello store impostiamo ***persist*** a true

counter.js

```
import { defineStore } from 'pinia'

export const useCounterStore = defineStore('counter', {
  state: () => ({
    count: 0
  }),
  getters: {
    doubleCount: (state) => state.count * 2,
    doubleCountPlusOne() { return this.doubleCount + 1 },
  },
  actions: {
    increment() {this.count++ },
  },
  persist: true,
})
```

pinia-plugin-persistedstate

- Impostando semplicemente *persist: true* nello store, il plugin utilizza una configurazione predefinita:
 - **localStorage** viene usato per salvare i dati
 - **store.\$id** come chiave di salvataggio dello store
 - **JSON.stringify/JSON.parse** per salvare e leggere i dati
 - L'intero contenuto di state dello store diventa persistente.
- Tuttavia, è possibile passare un oggetto alla proprietà *persist* dello store per configurare la persistenza.
- **key**, *tipo: string, default: store.\$id*, imposta il nome della chiave con cui salvare i dati nello storage.
- **storage**, *tipo: oggetto StorageLike. default: localStorage*, specifica dove salvare i dati: localStorage, sessionStorage o un tuo sistema personalizzato. Deve avere i metodi:
getItem: (key: string) => string | null
setItem: (key: string, value: string) => void.
- **paths**, *tipo: Array di string, default: undefined (tutto lo stato viene salvato)*, permette di salvare solo alcune parti dello stato. [] significa che nessun elemento di state è persistente.

Inviare richieste HTTP al server

- Fetch API fornisce un metodo globale **fetch()** che offre un modo semplice per **recuperare o inviare dati in modo asincrono** attraverso la rete.
- Il metodo **fetch()** consente di effettuare **richieste HTTP** (come GET, POST, ecc.) e restituisce una **Promise** che può essere gestita con **async/await**.
- Il risultato della richiesta è contenuto in un oggetto chiamato Response (comunemente abbreviato in **res**).
- La sintassi è **fetch(resource, options)**
 - **resource** è l'URL della risorsa che si vuole recuperare
 - **options** è un oggetto facoltativo che può specificare varie impostazioni, come il metodo HTTP (*methods*), gli *headers*, il corpo della richiesta (*body*), e altro ancora.
- Il tipo di richiesta più comune è la **GET**, usata **per recuperare dati** da un server.
- Vediamo un esempio di funzione da inserire in `methods`: per fare una richiesta di tipo GET inviando dei dati tramite `params`:

```
async searchSomething() {  
  const url = `url/to/${this.id}`  
  const res = await fetch(url)  
  if (res.status == 200) { var data = await res.json() }  
  else {alert("Qualcosa è andato storto durante la ricerca")}  
  .....  
}
```

```
data(){  
  return{  
    id : 0,  
  }  
},
```

Inviare richieste HTTP al server

- Le richieste **POST** sono invece utilizzate **per inviare dati** a un server. Vediamo come effettuare una richiesta POST con fetch()

```
methods: {  
  async sendFormData() {  
    const res = await fetch('url', {  
      method: 'POST',  
      headers: {  
        'Content-Type': 'application/json',  
      },  
      body: JSON.stringify(this.formData),  
    })  
    if (res.status !== 200) {  
      alert("Qualcosa è andato storto durante l'invio dei dati")  
      return;  
    }  
    const data = await res.json(); // Se ti serve la risposta del server  
    console.log('Risposta del server:', data);  
    ....  
  }  
}
```

```
data(){  
  return{  
    formData : {  
      username: 'username',  
      email: 'email',  
      password: 'xxxxx',  
    },  
  }  
},
```

Autenticazione

- Nello sviluppo web moderno, l'autenticazione è un elemento fondamentale per garantire la sicurezza delle applicazioni.
- L'autenticazione è il processo che verifica l'identità di un utente che tenta di accedere a una risorsa riservata. Può avvenire in diversi modi, ad esempio:
 - con **nome utente e password**;
 - tramite **social login** (Google, Facebook, ecc.);
 - con **autenticazione biometrica** (impronta digitale, riconoscimento facciale).
- I due approcci più diffusi sono:
 - l'autenticazione **stateful** (con memorizzazione lato server);
 - l'autenticazione **stateless** tramite **JSON Web Tokens (JWT)**.
- Nel modello **stateful**, il server mantiene informazioni di sessione dell'utente, salvandole ad esempio in un **database**.
- Nel modello **stateless**, il server non mantiene alcuno stato. L'autenticazione avviene tramite **JSON Web Token (JWT)**.
- Entrambi hanno vantaggi e svantaggi, e la scelta tra i due dipende dalle esigenze specifiche dell'applicazione.

Autenticazione stateful

- L'autenticazione basata sulla sessione (o **stateful**) è un metodo in cui il server memorizza i dati di autenticazione dell'utente, rendendoli persistenti per tutta la durata della sessione.
- Quando l'autenticazione ha successo (username e password corretti), il server **genera un token di sessione univoco**, che viene poi **salvato nel server**.
- Il token viene poi inviato al client (di solito come **cookie HTTP**).
- Ad ogni richiesta successiva, il client invia automaticamente il cookie al server, che **verifica il token** e autentica l'utente.

Pro e contro autenticazione stateful

Vantaggi

- **Facile da invalidare** - il server può terminare in qualsiasi momento una sessione sospetta, garantendo una maggiore sicurezza.
- **Miglior controllo delle informazioni sensibili** – alcune normative (GDPR) richiedono che i dati di sessione siano conservati sul server per garantire maggiore sicurezza, tutela della privacy e controllo in un ambiente protetto.
- **Autenticazione svolta una singola volta** – una volta autenticato, l'utente non ha bisogno di ripetere il login per ogni richiesta. Questo riduce il carico computazionale e migliora le prestazioni.

Svantaggi

- **Problemi di scalabilità** - con l'aumento del numero di utenti la memorizzazione di tutti i dati di sessione sul server può causare problemi di scalabilità.
- **Costo elevato** - richiede risorse e infrastrutture adeguate.
- **Complessità** - può essere complessa da implementare e mantenere.
- **Nessun accesso offline** - poiché tutti i dati della sessione sono memorizzati sul server.



Autenticazione stateful con Node.js

- `npm install express-session`
- ```
const express = require("express");
const session = require("express-session");
const app = express();
// Middleware per il parsing del corpo JSON
app.use(express.json());
// Middleware per la gestione della sessione
app.use(
 session({
 secret: "mysecretkey" // usato per firmare l'ID di sessione (evita manomissioni)
 resave: false, // evita di salvare la sessione se non è stata modificata
 saveUninitialized: false, // evita di salvare sessioni vuote
 cookie: {
 secure: true, // il cookie sarà inviato solo via HTTPS
 httpOnly: true, // impedisce l'accesso ai cookie da JavaScript lato client
 maxAge: 1000 * 60 * 60 // opzionale: durata del cookie in millisec (es. 1 ora)
 }
 })
);
```

Se stai sviluppando in locale senza HTTPS, le sessioni non funzioneranno correttamente se setti **secure:true** perché il cookie non viene mai inviato.

# Autenticazione stateful con Node.js

- Ogni sessione contiene:
  - **ID della sessione**, un identificatore univoco generato dal server e memorizzato nel browser dell'utente tramite un cookie.
  - **Dati di sessione**, qualsiasi informazione che si vuole conservare sul lato server e associare all'utente autenticato.
- Per ogni richiesta, Express crea un oggetto ***req.session***. È possibile aggiungere proprietà a questo oggetto per memorizzare dati legati alla sessione dell'utente.
- // Endpoint di login con salvataggio in sessione

```
app.post("/login", (req, res) => {
 const { username, password } = req.body;
 const user =; // Ricerca dell'utente nel DB usando username e password
 if (user) {
 req.session.userId = user.id; // Salva l'ID utente nella sessione
 res.json({ message: "Login effettuato con successo" });
 } else {
 // Se credenziali errate, invia errore 401 e un messaggio di errore
 res.status(401).json({ message: "Invalid username or password" });
 }
});
```

# Autenticazione stateful con Node.js

- Dopo il login, Express salverà automaticamente un cookie con ID di sessione nel browser dell'utente. Nelle richieste successive, sarà possibile accedere a ***req.session.userId*** per sapere quale utente è autenticato.
- ```
app.post("/logout", (req, res) => {  
  // Distrugge la sessione per disconnettere l'utente  
  req.session.destroy(err => {  
    if (err) {  
      // Se si verifica un errore nella distruzione della sessione  
      return res.status(500).json({ message: "Errore durante il logout" });  
    }  
    // Invia conferma al client  
    res.json({ message: "Logout effettuato con successo" });  
  });  
});
```
- Se il server si blocca o si riavvia, tutte le sessioni attive andranno perse.

Autenticazione stateless (JWT)

- L'autenticazione JWT è un metodo di autenticazione **stateless**, basato su token.
- A differenza dell'autenticazione tradizionale (stateful), **il server non memorizza informazioni di sessione.**
- Quando l'autenticazione ha successo (username e password corretti), il server genera un **JWT** che contiene informazioni sull'identità dell'utente.
- Il token è **firmato digitalmente** con una chiave segreta o privata e successivamente inviato al client, che lo **memorizza localmente** (es. utilizzando Pinia).
- In ogni richiesta successiva, il client **allega il token** (di solito nell'**header Authorization**).
- Il server **verifica la firma del token** per accertarsi che non sia stato alterato e per autenticare l'utente.

Pro e contro autenticazione stateless (JWT)

Vantaggi

- **Senza stato** - il token contiene tutte le informazioni necessarie per autenticare l'utente. Il server non ha bisogno di mantenere i dati di sessione o fare delle ricerche sul database per ogni richiesta. Questo semplifica la gestione e riduce il carico lato server.
- **Scalabilità** – i JWT facilitano la scalabilità perché il server non deve gestire o memorizzare lo stato delle sessioni.

Svantaggi

- **Dimensione del token** - i JWT possono diventare abbastanza grandi, soprattutto se contengono molte informazioni o se sono firmati con algoritmi robusti. Questo può rallentare le richieste, dato che il token viene trasmesso con ogni chiamata HTTP.
- **Rischi per la sicurezza** - se un token viene rubato o intercettato, un malintenzionato può utilizzarlo per accedere alle risorse come se fosse l'utente legittimo, finché il token resta valido, anche se firmato correttamente.
- **Scadenza del token** - se il token non scade, può essere utilizzato a tempo indeterminato. Tuttavia, se il token scade troppo frequentemente, può creare disagi agli utenti, che devono effettuare l'accesso frequentemente.

Autenticazione JWT con Node.js

- `npm install jsonwebtoken`
- ```
const express = require("express");
const jwt = require("jsonwebtoken");
const app = express();
const secretKey = "mysecretkey"; // Chiave segreta per firmare/verificare i token
app.use(express.json()); // Middleware per il parsing del corpo JSON
// Login endpoint
app.post("/login", (req, res) => {
 const { username, password } = req.body;
 const user =; // Ricerca dell'utente nel DB usando username e password
 if (user) {
 // Crea un token JWT con userId nel payload, valido per 1 ora
 const token = jwt.sign({ userId: user.id }, secretKey, { expiresIn: "1h" });
 res.json({ token }); // Invia il token al client
 } else { res.status(401).json({ message: "Nome utente o password non validi" }); }
});
```

# Autenticazione JWT con Node.js

- // Middleware riutilizzabile, separa la logica di verifica del token

```
function authenticateToken(req, res, next) {
 const authHeader = req.headers.authorization;

 if (!authHeader) {
 return res.status(401).json({ message: "Token mancante" });
 }

 // Estrae il token JWT dall'header (formato: "Bearer <token>")
 const token = authHeader.split(" ")[1];

 try {
 const decoded = jwt.verify(token, secretKey); // Verifica la validità del token
 req.userId = decoded.userId; /* Salva l'ID nel request object così le route
 successive possono accedervi facilmente */
 next(); // Passa al prossimo middleware/handler
 } catch (error) {
 return res.status(401).json({ message: "Token non valido o scaduto" });
 }
}
```



# Autenticazione JWT con Node.js

- // Endpoint protetto  
app.get("/profile", **authenticateToken**, (req, res) => {  
    const user = ..... /\* utilizza *req.userId* precedentemente estratto dal  
        JWT per trovare l'utente nel DB \*/  
  
    if (user) {  
        res.json({ user });  
    } else {  
        res.status(404).json({ message: "Utente non trovato" });  
    }  
});