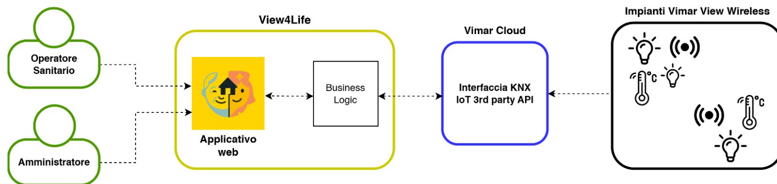
Presentazione architettura

Gruppo 1 - A.A 2025/2026

20/04/2026

Obiettivo capitolato

- Capitolato C9 (Vimar View4Life): Piattaforma per la gestione degli impianti Smart nelle residenze protette;
- Sistema di gestione allarmi e utenti (OSS e Amministrazione) che possono interagire con gli allarmi (gestione e risoluzione);
- Visualizzazione di statistiche sui consumi, con relativi consigli per ridurli, e sugli allarmi.



Architettura di deployment

- Frontend: *Single Page Application* in Angular, servita da *nginx*;
- Backend: monolite realizzato con NestJS.

Entrambi, come il database *PostgreSQL*, sono containerizzati e orchestrati da *Docker Compose*.

Attualmente tutti e tre i container sono in esecuzione in un singolo nodo.

Comunicazione tra *frontend* e *backend*

La comunicazione tra *frontend* e *backend* avviene tramite *API REST*.

AlarmRules

Metodo	Route
POST	/alarm-rules
GET	/alarm-rules
GET	/alarm-rules/{id}
PUT	/alarm-rules/{id}
DELETE	/alarm-rules/{id}

Figura: *route* per le regole di allarme

AlarmEvents

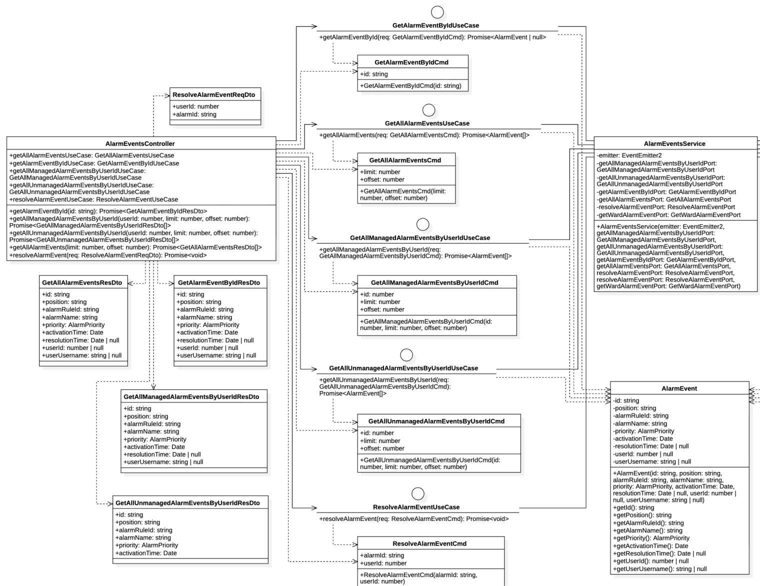
Metodo	Route
GET	/alarm-events/{id}
GET	/alarm-events/managed/{userId}/{limit}/{offset}
GET	/alarm-events/unmanaged/{userId}/{limit}/{offset}
GET	/alarm-events/{limit}/{offset}
PATCH	/alarm-events/resolve

Figura: *route* per gli eventi di allarme

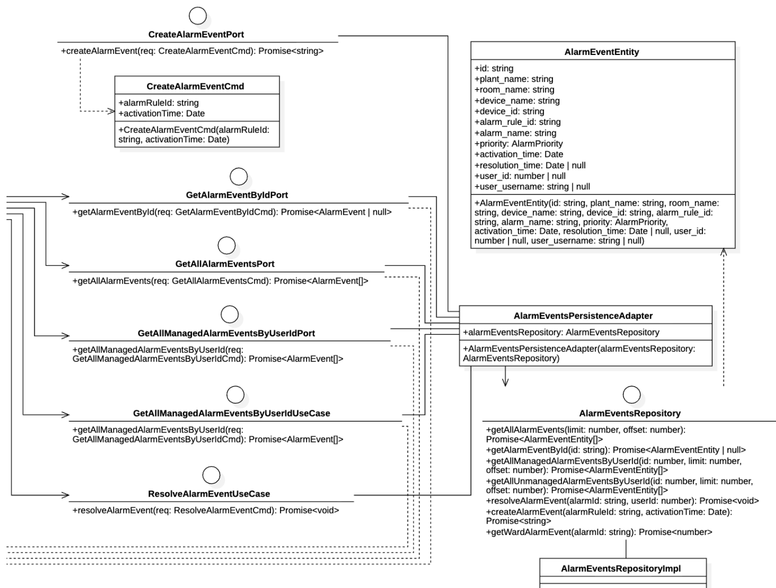
Architettura logica *backend*

- Problema:
  - La logica di *business* del *backend* dipende dai dettagli infrastrutturali (*Persistent Logic*) e dalla tecnologia scelta per la codifica;
  - Difficile testare il *backend* senza dover istanziare l'intera infrastruttura.
- Soluzione: **Architettura Esagonale**
  - La logica di *business* è isolata dalle altre componenti;
  - Tramite le **Ports** e **Adapters**, situati ai confini della logica di *business*, è possibile interfacciarsi con essa;
  - La *business* logic può essere verificata in modo indipendente.

UML Alarm (Backend) (1/2)



UML Alarm (Backend) (2/2)

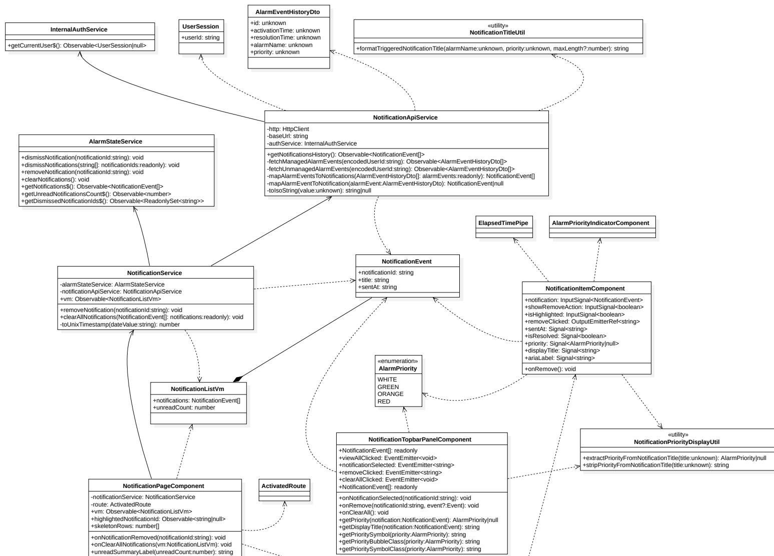


SnakeByte

Architettura logica *frontend*

- Problema:
  - Logica di presentazione legata all'interfaccia utente;
  - Ogni cambiamento della vista rischia di impattare la logica di business.
- Soluzione: **Architettura MVVM** che separa le tre responsabilità:
  - **Model**: dati e logica di dominio, implementati come interfacce/tipi e come servizi (`@Injectable`).
  - **View**: i template HTML del componente; passiva, si limita a realizzare le viste mostrando i dati via **property binding** e a notificare eventi via **event binding**.
  - **ViewModel**: la classe del componente (`@Component`); fa da mediatore tra Model e View, esponendo i dati tramite `Observable` senza contenere logica di dominio.

UML Notification (Frontend)



Code snippets Notification (Frontend)

```typescript
@Component({
  selector: 'app-notification-page-component',
  changeDetection: ChangeDetectionStrategy.OnPush,
  imports: [AsyncPipe, NotificationItemComponent],
  providers: [NotificationService],
  templateUrl: './notification-page-component.html',
  styleUrl: './notification-page-component.css',
})
export class NotificationPageComponent {
  private readonly notificationService = inject(NotificationService);
  private readonly route = inject(ActivatedRoute);

  public readonly vm$: Observable<NotificationListVm> = this.notificationService.vm$;

  public onClearAllNotifications(vm: NotificationListVm): void {
    this.notificationService.clearAllNotifications(vm.notifications);
  }
```

Figura: notification-page-component.ts

```html
@if (vm$ | async; as vm) {
<div class="notification-list space-y-4">
    <div class="flex w-full flex-wrap items-center justify-between gap-3">
        <span
            class="notification-page__summary inline-flex items-center gap-2">
            <span class="h-2 w-2 rounded-full bg-amber-500"></span>
            {{ unreadSummaryLabel(vm.unreadCount) }}
        </span>

        @if (vm.notifications.length > 0) {
        <button
            type="button"
            class="ml-auto cursor-pointer inline-flex items-center gap-2"
            (click)="onClearAllNotifications(vm)">
            Cancella tutte
        </button>
        }
    </div>
```

Figura: notification-page-component.html

Design Pattern - Dependency Injection (1/2)

- Problema:
  - I moduli sono fortemente accoppiati tra di loro;
  - Il codice risulta difficile da testare e da manutenere.
- Soluzione: pattern **Dependency Injection**
  - Le dipendenze vengono fornite tramite un `INJECTOR`, che si occupa di creare e gestire il ciclo di vita degli oggetti al posto delle classi stesse;
  - Il pattern è supportato nativamente in entrambi i *framework* tramite il decoratore `@Injectable()`: in *NestJS* il *binding* è esplicito tramite il sistema `providers`, mentre in *Angular* i service si registrano automaticamente nel container tramite `providedIn:'root'`.

Design Pattern - Dependency Injection (2/2)

```
providers: [
    { provide: CREATE_ALARM_RULE_USE_CASE, useClass: AlarmRulesService },
    { provide: CREATE_ALARM_RULE_PORT, useClass: AlarmRulesPersistenceAdapter },
    { provide: DELETE_ALARM_RULE_USE_CASE, useClass: AlarmRulesService },
    { provide: DELETE_ALARM_RULE_PORT, useClass: AlarmRulesPersistenceAdapter },
    { provide: GET_ALL_ALARM_RULES_USE_CASE, useClass: AlarmRulesService },
```

Figura: Sistema *providers* di *NestJS* per la D.I.

```
@Injectable()
export class AlarmRulesService
  implements
    CreateAlarmRuleUseCase,
    GetAlarmRuleByIdUseCase,
    GetAllAlarmRulesUseCase,
    UpdateAlarmRuleUseCase,
    DeleteAlarmRuleUseCase,
    CheckAlarmRuleUseCase
{
```

Figura: Sistema *@Injectable* di *NestJS* per la D.I.

```
@Controller('alarm-rules')
export class AlarmRulesController {
  constructor(
    @Inject(CREATE_ALARM_RULE_USE_CASE)
    private readonly createAlarmUseCase: CreateAlarmRuleUseCase,
    @Inject(DELETE_ALARM_RULE_USE_CASE)
    private readonly deleteAlarmRuleUseCase: DeleteAlarmRuleUseCase,
    @Inject(GET_ALARM_RULE_BY_ID_USE_CASE)
    private readonly getAlarmRuleByIdUseCase: GetAlarmRuleByIdUseCase,
    @Inject(GET_ALL_ALARM_RULES_USE_CASE)
    private readonly getAllAlarmRulesUseCase: GetAllAlarmRulesUseCase,
    @Inject(UPDATE_ALARM_RULE_USE_CASE)
    private readonly updateAlarmRuleUseCase: UpdateAlarmRuleUseCase,
  ) {}
```

Figura: Sistema *@Inject* di *NestJS* per la D.I.

Design Pattern - Observer (1/3)

- Problema:
  - I componenti *Angular* devono reagire a eventi asincroni e aggiornare lo stato *UI*.
- Soluzione: design pattern **Observer**
  - Implementato tramite libreria *RxJS*: le entità si iscrivono agli *Observable* senza conoscere la sorgente dei dati;
  - Gli *Observable* gestiscono eventi in maniera automatica ed efficiente mantenendo disaccoppiamento tra le parti coinvolte.

Design Pattern - Observer (2/3)

ChartDatasetDto
+id: string
+name: string
+data: number[]

EnergySavingSuggestionDto
+messages: string[]
+isSuggestion: boolean

ChartInfoDto
+labels: string[]
+datasets: ChartDatasetDto[]
+title: string
+unit: string
+suggestions: EnergySavingSuggestionDto
+metric: string

AnalyticsDto
+apartmentId: string
+analyticsInfo: ChartInfoDto[]

«abstract»
ChartComponent
+chartInfo: ChartInfoDto
+chartType: ChartType
+renderedChartData: ChartData
+chartOptions: ChartOptions
+chartColors: object[]
+showSuggestions: boolean
+ngOnChanges(changes:SimpleChanges): void
#updateChart(data:ChartInfoDto): void
+showHideSugg(): void

AlarmFrequencyChartComponent
+chartType: ChartType

AlarmsSentResolvedChartComponent
+chartType: ChartType

EnergyConsumptionChartComponent
+chartType: ChartType
+chartColors: object[]

FallFrequencyChartComponent
+chartType: ChartType

PlantAnomaliesChartComponent
+chartType: ChartType

PresenceDetectionChartComponent
+chartType: ChartType

ProlongedPresenceChartComponent
+chartType: ChartType

TemperatureVariationsChartComponent
+chartType: ChartType
+chartColors: object[]

AnalyticsApiService
-http: HttpClient
-baseUrl: string
-analyticsEndpoint: string
-apartmentsEndpoint: string
+getAllApartments(): Observable<any[]>
+getAnalytics(apartmentId:string, refresh?:boolean): Observable<AnalyticsDto>

AnalyticsComponent
-refreshTrigger: BehaviorSubject<boolean>
-analyticsApiService: AnalyticsApiService
-loadTrigger: BehaviorSubject<object|null>
+selectedApartmentId: BehaviorSubject<string|null>
+apartments: Observable<any[]>|null
+analytics: Observable<AnalyticsDto|null>|null
+ngOnInit(): void
+getChartByMetric(data:AnalyticsDto|null, metric:string): ChartInfoDto|undefined
+onApartmentChange(event:Event): void
+onRefresh(): void

Design Pattern - Observer (3/3)

```
1  export class AnalyticsComponent implements OnInit {
2
3      private readonly analyticsApiService = inject(AnalyticsApiService);
4      private readonly loadTrigger$ = new BehaviorSubject<{ id: string; refresh: boolean } | null>(null);
5
6      public selectedApartmentId$ = new BehaviorSubject<string | null>(null);
7
8      public apartments$: Observable<any[]> | null = null;
9      public analytics: Observable<AnalyticsDto | null> | null = null;
```

Figura: analytics.component.ts

```
1  <div *ngIf="analytics | async as data; else loading"
2  class="grid grid-cols-6 w-full gap-8 content-start">
3
4      <app-energy-consumption-chart
5          *ngIf="getChartByMetric(data, 'plant-consumption') as info"
6          class="col-span-6 2xl:col-span-3 w-full"
7          [chartInfo]="info">
8      </app-energy-consumption-chart>
9      ...
```

Figura: Iscrizione all'interno del template tramite `async pipe`

Design Pattern - Strategy (1/2)

- Problema:
  - Visualizzare un'insieme di *analytics* relative agli impianti;
  - Ogni *analytics* ha un grafico specifico, basato su dati differenti (Es. consumo della luce, temperatura media del termostato, ecc.);
  - Deve essere possibile modificare le *analytics* se necessario.
- Soluzione: **Pattern Strategy**
  - È stata realizzata un'interfaccia `AnalyticsStrategy` che fornisce il metodo `execute`, che le *strategies* implementano.

Design Pattern - Strategy (2/2)