Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Лабораторна робота №9

**Тема:** Взаємодія компонентів системи.

Виконала
студентка групи ІА–32:
Слюсарева А.А.

Київ 2025

**ЗМІСТ**

# ЛАБОРАТОРНА РОБОТА № 9

**Тема:** Взаємодія компонентів системи.

**Мета:** Вивчити види взаємодії додатків (Client-Server, Peer-to-Peer, Service oriented Architecture), та реалізувати в проєктованій системі одну із архітектур.

## 9.1. Завдання

• Ознайомитись з короткими теоретичними відомостями.

• Реалізувати частину функціоналу робочої програми у вигляді класів та їхньої взаємодії для досягнення конкретних функціональних можливостей.

• Реалізувати функціонал для роботи в розподіленому оточенні відповідно до обраної теми.

• Реалізувати взаємодію розподілених частин:

– Для клієнт-серверних варіантів: реалізація клієнтської і серверної частини додатків, а також загальної частини (middleware); зв'язок клієнтської і серверної частин за допомогою WCF, TcpClient, .NET Remoting на розсуд виконавця.

– Для однорангових мереж: реалізація взаємодії клієнтських додатків за допомогою WCF Peer to peer channel.

– Для SOA додатків: реалізація сервісу, що надає послуги клієнтським застосуванням; викладання сервісу в хмару або підняття у вигляді Web Service на локальній машині; використання токенів для передачі даних про автентифікації, двостороннє шифрування.

• Підготувати звіт щодо виконання лабораторної роботи. Поданий звіт повинен містити: діаграму класів, яка представляє спроєктовану архітектуру. Навести фрагменти програмного коду, які є суттєвими для відображення реалізованої архітектури.

## 9.2. Теоретичні відомості

### 9.2.1. Клієнт-серверна архітектура

Клієнт-серверна архітектура - це модель розподілених систем, у якій виділяють два типи додатків:
клієнти (відображають інтерфейс та приймають дії користувача) і сервери (зберігають та обробляють дані).

Тонкий клієнт:

- Основна логіка обробки даних виконується на сервері.
- Клієнт переважно відповідає за інтерфейс та відправку запитів.
- Приклад: класичні веб-додатки.
- Переваги: просте розгортання — оновлення потрібне лише на сервері.

Товстий клієнт:

- Значна частина логіки виконується на стороні клієнта.
- Сервер часто лише забезпечує доступ до даних.
- Можлива робота без підключення до серверу.
- Приклади: мобільні додатки, десктоп-програми (Evernote, Viber, Outlook, ігри).
- Переваги: менше навантаження на сервер.

Проміжний варіант - SPA (Single Page Application):

- Товстий веб-клієнт, який завантажується з сервера один раз, а далі працює через Web API.
- Менше навантаження на сервер; оновлення простіше, ніж у звичайних товстих клієнтів.
- Не працює без доступу до сервера.

Типова трирівнева структура:

1. Клієнтська частина - інтерфейс, логіка взаємодії з користувачем, виклики до сервера.
2. Загальна частина (middleware) - спільні моделі, типи даних, інструменти.
3. Серверна частина - бізнес-логіка, обробка даних, взаємодія між компонентами.

### 9.2.2. Peer-to-Peer (P2P) архітектура

P2P - це децентралізована модель, у якій кожен вузол одночасно виступає і клієнтом, і сервером. Усі учасники взаємодіють напряму, без центрального сервера.

Основні принципи:

- Децентралізація - немає єдиного сервера; мережа стійкіша до збоїв.
- Рівноправність вузлів - кожен може надавати й отримувати ресурси.
- Розподіл ресурсів - дисковий простір, процесорний час, файли тощо.

Сфери застосування:

- Файлообмінники (BitTorrent)
- Криптовалюти та блокчейн
- Інтернет-телефонія (Skype, Zoom)

- Розподілені обчислення (SETI@home, BOINC)

Проблемні зони:

- Безпека й контроль передаваних даних
- Складність синхронізації та узгодженості
- Пошук ресурсів у великих мережах вимагає спеціальних алгоритмів

### 9.2.3. Сервіс-орієнтована архітектура (SOA)

SOA - це модульний підхід, у якому система складається з незалежних сервісів зі стандартизованими інтерфейсами. Сервіси слабко пов’язані та взаємодіють через стандартизовані протоколи.

Ключові риси:

- Сервіси зазвичай реалізовані як веб-служби (SOAP або REST).
- Кожен сервіс виконує чітко визначену бізнес-функцію.
- Взаємодія здійснюється через обмін повідомленнями, а не через спільну базу даних.
- Старі системи можна інтегрувати за допомогою «обгорток» (adapters).
- Часто використовується центральна шина даних (ESB).

Переваги:

- Легше інтегрувати різні системи.
- Можливість незалежної розробки та повторного використання сервісів.
- Гнучкість і масштабованість у порівнянні з монолітною архітектурою.

### 9.2.4. Мікросервісна архітектура

Мікросервіси - подальший розвиток SOA, орієнтований на малі, незалежні служби, кожна з яких виконується у власному процесі та має власний життєвий цикл.

Властивості мікросервісів:

- Кожен мікросервіс відповідає за конкретну бізнес-можливість.
- Розробляється й розгортається незалежно від інших.
- Взаємодія відбувається через HTTP/HTTPS, WebSockets, AMQP або інші протоколи.
- Може бути створений будь-якою мовою та мати власну базу даних.

Визначення (за O’Reilly):

Мікросервіс - це компонент із чіткими межами, який можна розгортати незалежно і який взаємодіє з іншими через обмін повідомленнями.

Переваги:

- Висока масштабованість
- Зручність супроводження великих систем
- Гнучке розгортання і оновлення без зупинки всієї системи

## 9.3 Хід роботи

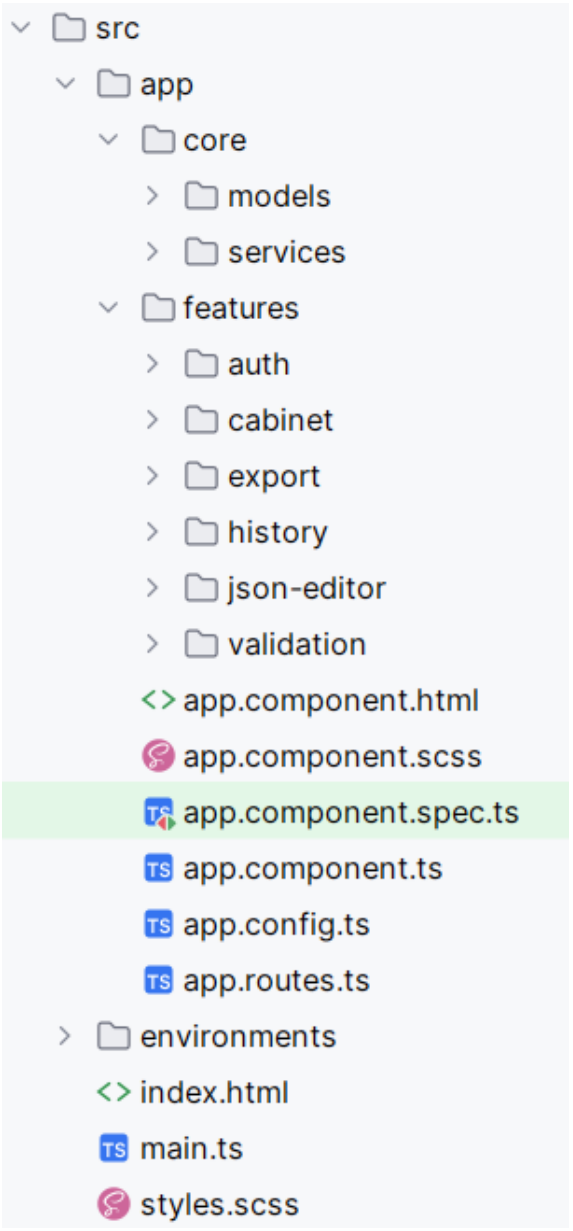


Рис 9.3.1 Структура коду проєкта

```
@Injectable({  Show usages  Anastasiia Sliusareva
  providedIn: 'root'
})
export class FirebaseService {
  private app: FirebaseApp;
  private auth: Auth;
  private firestore: Firestore;
  private currentUserSubject: BehaviorSubject<User | null>;
  public currentUser$: Observable<User | null>;

  constructor() {  no usages  Anastasiia Sliusareva
    this.app = initializeApp(environment.firebase);
    this.auth = getAuth(this.app);
    this.firestore = getFirestore(this.app);
    this.currentUserSubject = new BehaviorSubject<User | null>( _value: null);
    this.currentUser$ = this.currentUserSubject.asObservable();

    onAuthStateChanged(this.auth, (user : User | null ) : void  => {
      this.currentUserSubject.next(user);
    });
  }

  get currentUser(): User | null {  Show usages  Anastasiia Sliusareva
    return this.currentUserSubject.value;
  }

  async signInWithEmail(email: string, password: string): Promise<User> {  Show usages  Anastasiia Sliusareva
    const result : UserCredential  = await signInWithEmailAndPassword(this.auth, email, password);
    return result.user;
  }
```

Рис 9.3.2 Код API проєкту(ч.1)

```
async signUpWithEmail(email: string, password: string): Promise<User> {  Show usages  Anastasiia Sliusareva
  const result : UserCredential  = await createUserWithEmailAndPassword(this.auth, email, password);
  return result.user;
}

async signInWithGoogle(): Promise<User> {  Show usages  Anastasiia Sliusareva
  const provider = new GoogleAuthProvider();
  const result : UserCredential  = await signInWithPopup(this.auth, provider);
  return result.user;
}

async signOut(): Promise<void> {  Show usages  Anastasiia Sliusareva
  await signOut(this.auth);
}

async saveUserSchema(userId: string, schemaId: string, schemaData: any, jsonText?: string, name?: string): Promise<void> {  Show usages
  const schemaRef : DocumentReference<DocumentData, DocumentD...  = doc(this.firestore,  path: `users/${userId}/schemas/${schemaId}`);
  await setDoc(schemaRef, {
    ...schemaData,
    jsonText: jsonText ?? '',
    name: name ?? schemaId,
    updatedAt: new Date().toISOString()
  });
}

async updateSchemaName(userId: string, schemaId: string, name: string): Promise<void> {  Show usages  Anastasiia Sliusareva
  const schemaRef : DocumentReference<DocumentData, DocumentD...  = doc(this.firestore,  path: `users/${userId}/schemas/${schemaId}`);
  const schemaDoc : DocumentSnapshot<DocumentData, DocumentDa...  = await getDoc(schemaRef);
  if (schemaDoc.exists()) {
```

Рис 9.3.3 Код API проєкту(ч.2)

```
  async updateSchemaName(userId: string, schemaId: string, name: string): Promise<void> {  Show usages  Anastasiia Sliusareva
    const schemaRef : DocumentReference<DocumentData, DocumentD... = doc(this.firestore, path: `users/${userId}/schemas/${schemaId}`);
    const schemaDoc : DocumentSnapshot<DocumentData, DocumentDa... = await getDoc(schemaRef);
    if (schemaDoc.exists()) {
      await setDoc(schemaRef, {
        ...schemaDoc.data(),
        name,
        updatedAt: new Date().toISOString()
      });
    }
  }

  async getUserSchema(userId: string, schemaId: string): Promise<any> {  Show usages  Anastasiia Sliusareva
    const schemaRef = doc(this.firestore, `users/${userId}/schemas/${schemaId}`);
    const schemaDoc = await getDoc(schemaRef);
    return schemaDoc.exists() ? schemaDoc.data() : null;
  }

  async getUserSchemas(userId: string): Promise<any[]> {  Show usages  Anastasiia Sliusareva
    const schemasRef = collection(this.firestore, `users/${userId}/schemas`);
    const snapshot = await getDocs(schemasRef);
    return snapshot.docs.map(doc => ({ id: doc.id, ...doc.data() }));
  }

  async deleteUserSchema(userId: string, schemaId: string): Promise<void> {  Show usages  Anastasiia Sliusareva
    const schemaRef = doc(this.firestore, `users/${userId}/schemas/${schemaId}`);
    await deleteDoc(schemaRef);
  }
}
```

Рис 9.3.4 Код API проєкту(ч.3)

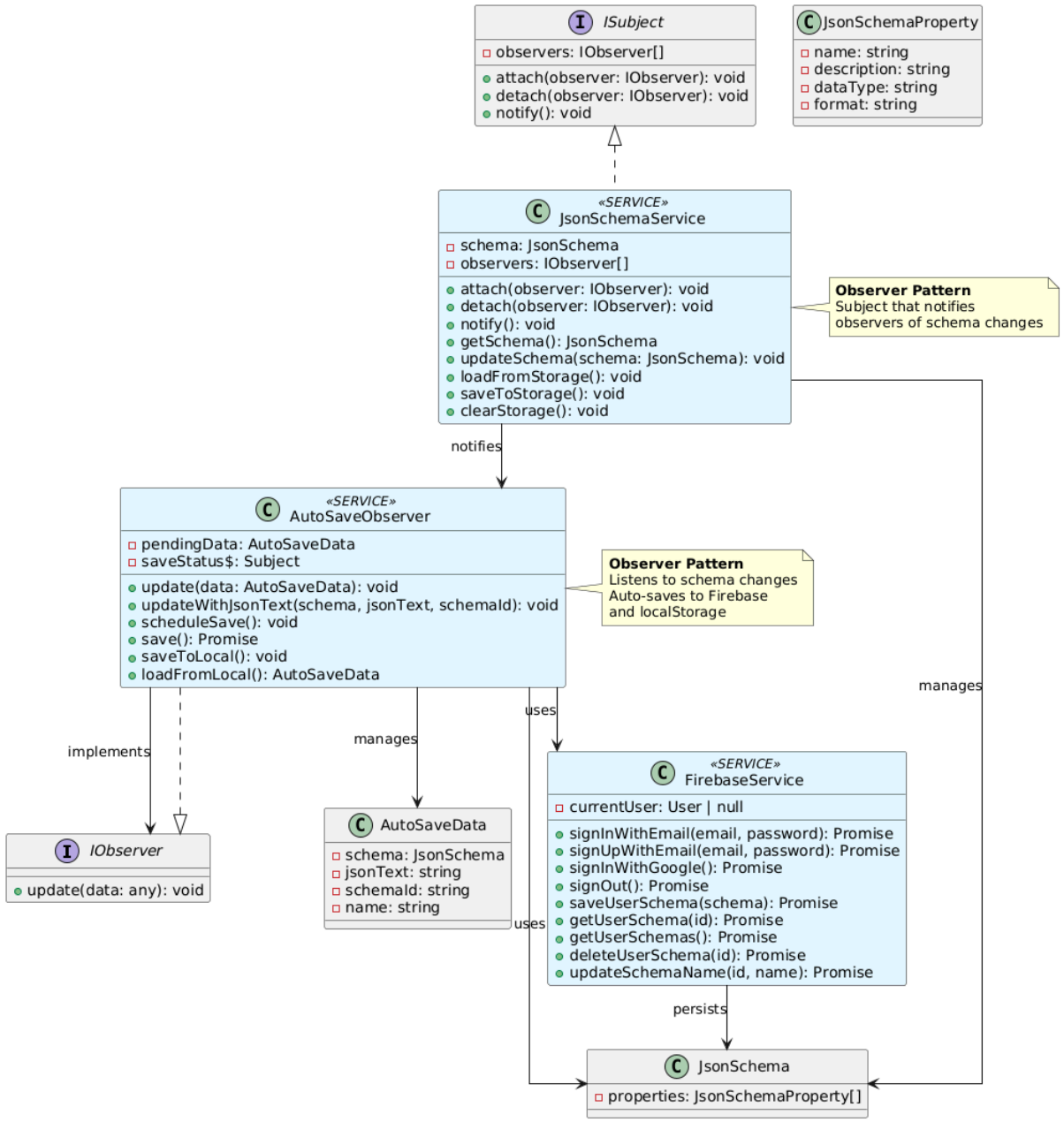


Рис 9.3.5 Архітектура базових сервісів

Діаграма показує реалізацію паттерну Observer у системі управління схемами JSON. Система побудована таким чином, що будь-які зміни схеми автоматично повідомляються зацікавленим компонентам без прямого зв'язування.

Основні компоненти:

JsonSchemaService — центральний сервіс управління станом

- Зберігає поточну схему JSON
- Реалізує інтерфейс ISubject паттерну Observer
- Керує списком спостерігачів (observers)
- Методи: attach(), detach(), notify() для управління спостерігачами
- Методи: updateSchema() для змін схеми, loadFromStorage()/saveToStorage() для локального сховища

AutoSaveObserver — спостерігач для автоматичного збереження

- Реалізує інтерфейс IObserver
- Слухає сигнали від JsonSchemaService через метод update()
- Керує чергою автозбереження з затримкою (debounce)
- Синхронізує дані в Firebase та localStorage одночасно
- Має статус сохранения для відображення користувачеві

FirebaseService — сервіс для роботи з хмарою

- Забезпечує аутентифікацію користувача
- Зберігає схеми в Firestore базі даних
- Методи для CRUD операцій: saveUserSchema(), getUserSchema(), deleteUserSchema()

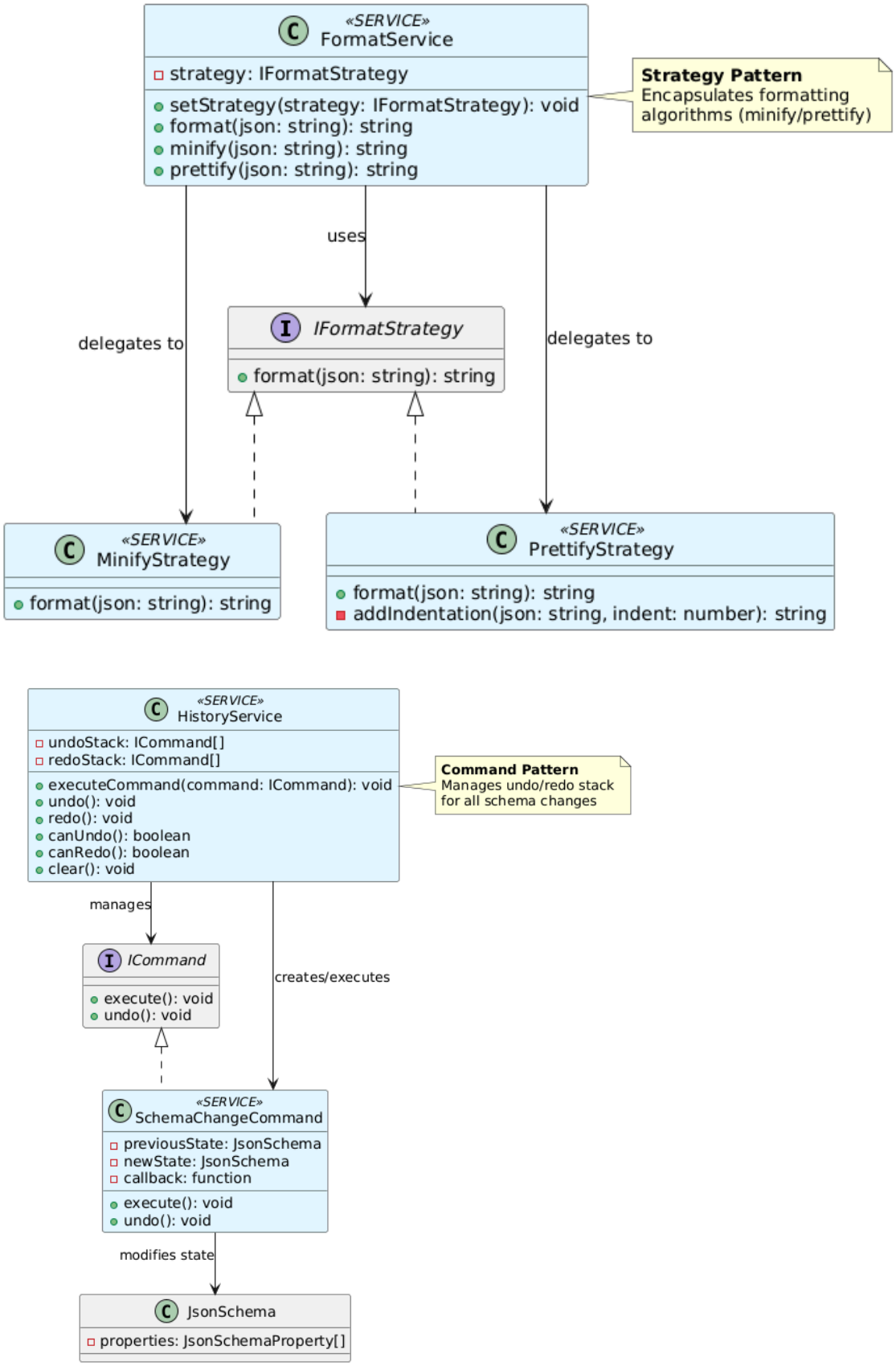


Рис 9.3.6 Сервіси форматування та історії

Діаграма демонструє два критичні патерни дизайну: Strategy для гнучкого форматування та Command для функціональності undo/redo.

Основні компоненти:

FormatService — сервіс форматування з гнучкою архітектурою

- Реалізує паттерн Strategy для заміни алгоритмів в runtime
- Держить поточну стратегію форматування
- Методи: setStrategy() для вибору алгоритму, format() для застосування
- Зручні методи: minify() і prettify() без явного вибору стратегії

Стратегії форматування

- MinifyStrategy — видаляє пробіли, перевідства рядків (зменшує розмір)
- PrettifyStrategy — додає відступи, переводи рядків (для читаності)
- Обидві реалізують інтерфейс IFormatStrategy

HistoryService — управління undo/redo функціональністю

- Реалізує паттерн Command для інкапсуляції змін
- Керує двома стеками: undoStack та redoStack
- Методи: executeCommand() для виконання команди, undo(), redo() для навігації
- Методи перевірки: canUndo() та canRedo() для активації кнопок в UI

SchemaChangeCommand — команда для зміни схеми

- Зберігає попередній та новий стан схеми
- Методи: execute() для виконання зміни, undo() для повернення до попереднього стану
- Дозволяє будь-яку операцію зробити відмінною та повторюваною

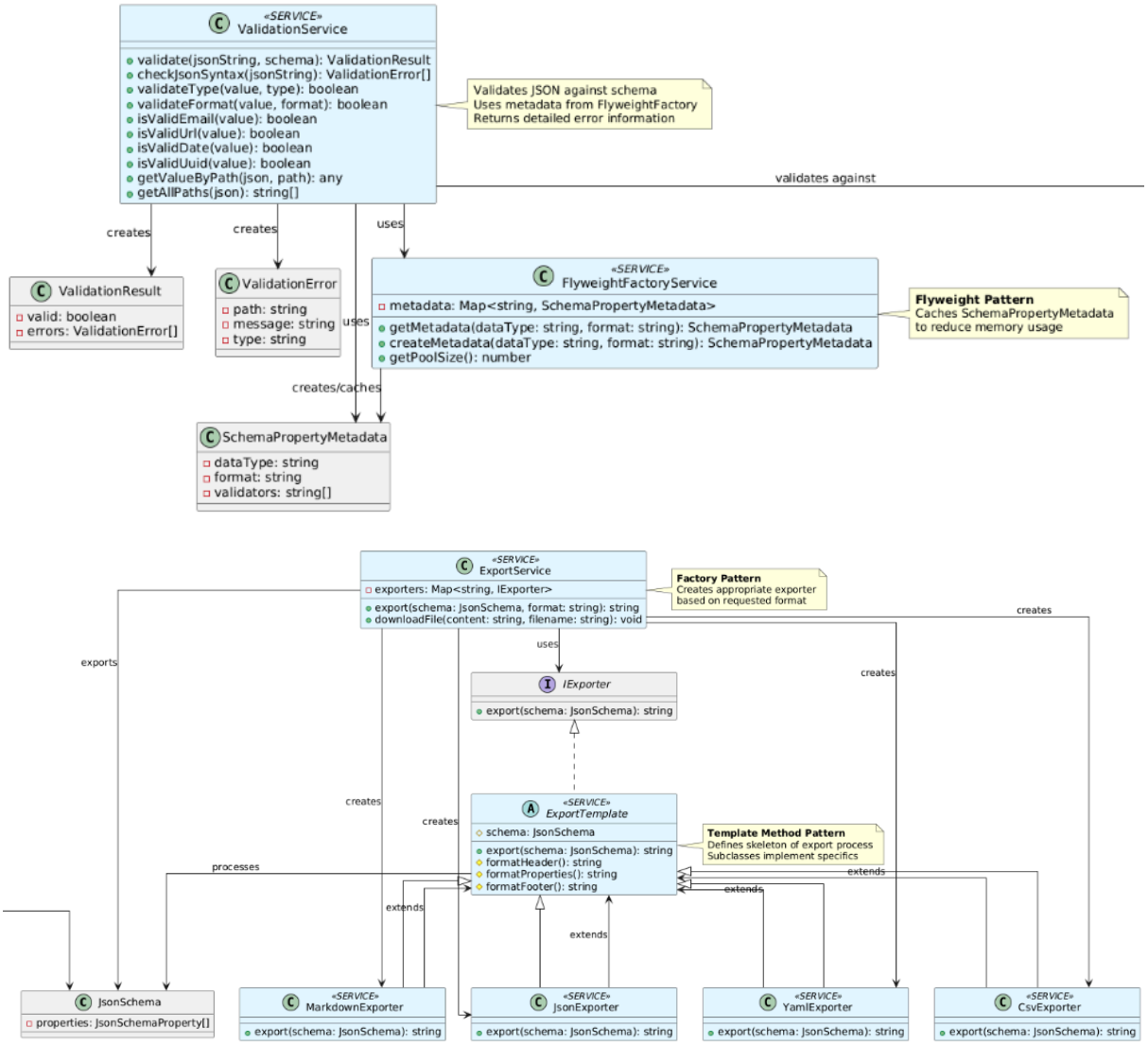


Рис 9.3.7 Валідація та експорт

Діаграма показує два комплекси функціональності: валідацію JSON проти схеми та експорт схем у різні формати. Використовуються паттерни Flyweight для оптимізації пам'яті, Factory для створення експортерів та Template Method для консистентної логіки.

Основні компоненти:

ValidationService — валідація JSON

- Перевіряє JSON на відповідність схемі
- Методи для перевірки: синтаксис, типи даних, формати (email, URL, UUID, дата)
- Утилітарні методи: getValueByPath() для пошуку значення по шляху, getAllPaths() для отримання всіх ключів
- Повертає детальний ValidationResult з переліком помилок

FlyweightFactoryService — кешування метаданих

- Реалізує паттерн Flyweight для економії пам'яті
- Зберігає кеш метаданих для кожної комбінації (dataType + format)
- При повторному запиту повертає той же об'єкт з кешу
- Метод getPoolSize() показує кількість кешованих елементів

ExportService — фабрика експортерів

- Реалізує паттерн Factory для створення експортерів
- Зберігає map експортерів по форматам
- Метод export() вибирає правильний експортер за форматом
- Метод downloadFile() допомагає завантажити файл браузером

ExportTemplate — абстрактний клас з шаблоном

- Реалізує паттерн Template Method
- Визначає кістяк процесу експорту: formatHeader() → formatProperties() → formatFooter()
- Підкласи переозначують методи форматування для своїх форматів

Конкретні експортери

- MarkdownExporter — експортує в Markdown таблиці
- JsonExporter — експортує у JSON формат
- YamlExporter — експортує у YAML формат
- CsvExporter — експортує у CSV формат

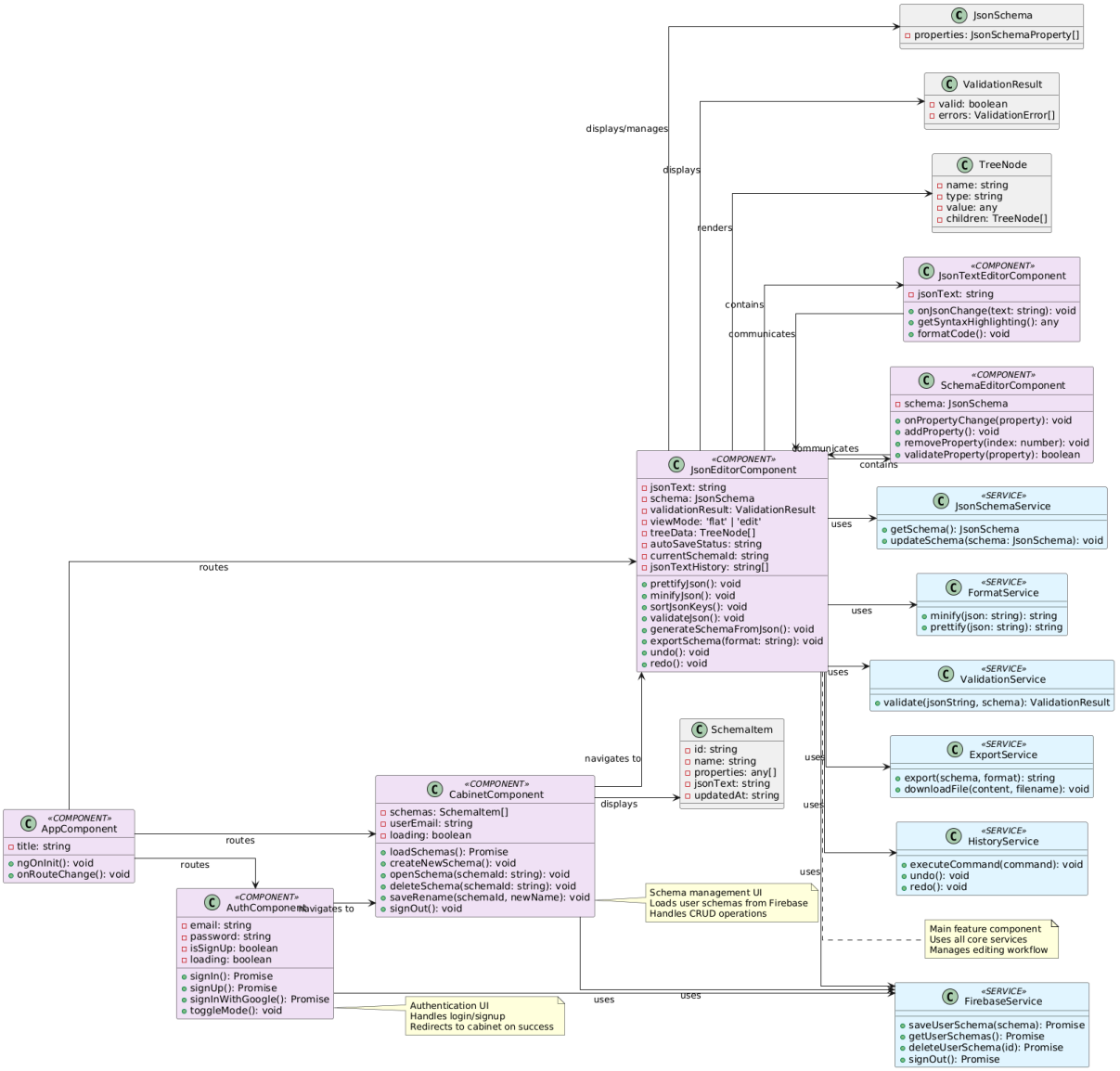


Рис 9.3.8 Архітектура компонентів(класів)

Діаграма показує структуру Angular компонентів та їх залежності від сервісів. Система розділена на три функціональні області: аутентифікація, управління схемами в кабінеті та основний редактор.

Основні компоненти:

AppComponent — коренева компонента

- Налаштовує маршрутизацію між основними сторінками
- Точка входу до всієї системи

AuthComponent — аутентифікація

- Сторінка входу та реєстрації користувача
- Методи: signIn(), signUp(), signInWithGoogle()
- Використовує FirebaseService для аутентифікації
- Перенаправляє на CabinetComponent при успішному вході

CabinetComponent — кабінет користувача

- Відображає список схем, збережених користувачем
- Методи для роботи зі схемами: loadSchemas(), createNewSchema(), openSchema(), deleteSchema()
- Використовує FirebaseService для завантаження схем
- Дозволяє вибрати схему для редагування або створити нову

JsonEditorComponent — головний редактор

- Основна компонента для редагування JSON та схем

Властивості:

- jsonText — поточний JSON текст
- schema — поточна схема
- validationResult — результати валідації
- viewMode — режим перегляду (flat чи edit)
- jsonTextHistory — історія для undo/redo
- Використовує всі сервіси системи:
- JsonSchemaService — управління схемою
- FormatService — форматування JSON
- ValidationService — валідація
- ExportService — експорт схеми
- HistoryService — undo/redo
- FirebaseService — синхронізація
- Методи: prettifyJson(), minifyJson(), validateJson(), exportSchema(), undo(), redo()

JsonTextEditorComponent — редактор JSON тексту

- Підкомпонента JsonEditorComponent

- Забезпечує синтаксичне підсвічування та форматування
- Інтеграція з батьківської компонентом через @Input/@Output

SchemaEditorComponent — редактор схеми

- Підкомпонента JsonEditorComponent
- Керування властивостями схеми
- Додавання, видалення та редагування властивостей
- Валідація властивостей перед збереженням

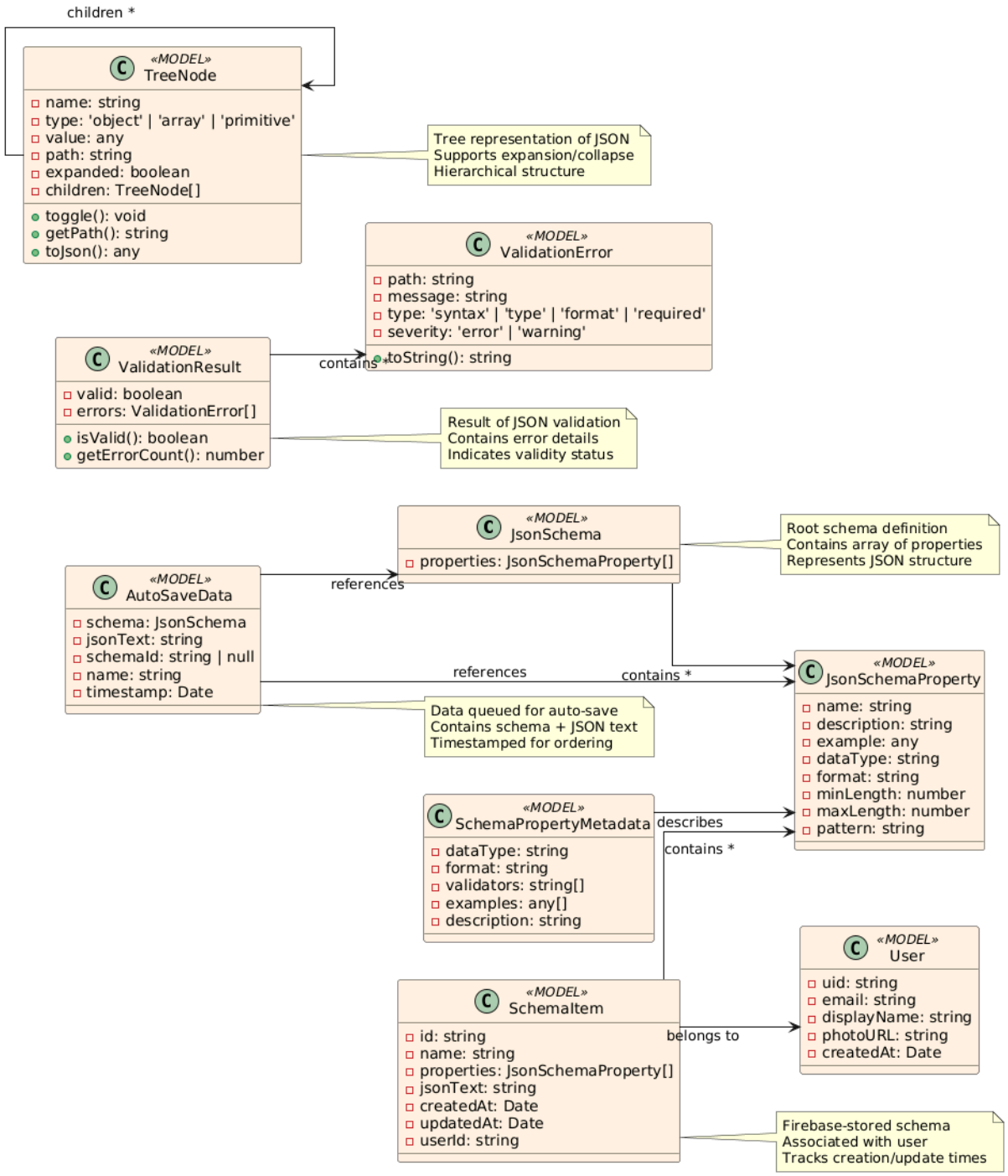


Рис 9.3.9 Моделі даних та DTO

Діаграма показує все дані, що циркулюють у системі. Дані розділені на 6 категорій, кожна з яких використовується для конкретної мети.

Основні моделі:

Моделі схем:

- JsonSchema — коренева схема, містить масив властивостей
- JsonSchemaProperty — окрема властивість з типом, форматом, прикладами, валідаторами

Моделі валідації

- ValidationResult — результат валідації з прапором valid та переліком помилок
- ValidationError — деталь про помилку: шлях до помилки, повідомлення, тип

Моделі автозбереження

- AutoSaveData — упакована дата для автозбереження (схема + JSON текст + метадані + timestamp)
- Моделі Firebase
- SchemaItem — схема, збережена в базі, з ID, назвою, часом створення/оновлення
- User — користувач з UID, email, фото, датою створення

Моделі представлення

- TreeNode — деревна структура JSON для інтерактивного відображення у UI
- Підтримує розширення/згортання гілок
- Рекурсивна структура (діти також TreeNode)

Моделі метаданих

- SchemaPropertyMetadata — кешовані метадані про властивість (валідатори, приклади, описи)

### 9.4 Висновок

В ході виконання цієї лабораторної роботи було вивчено види архітектур систем і використано ці знання на підготовленому проєкті,

## 9.5. Питання до лабораторної роботи

1. **Що таке клієнт-серверна архітектура?**
   Модель, у якій клієнт запитує ресурси чи послуги, а сервер їх обробляє та повертає результат.

2. **Розкажіть про сервіс-орієнтовану архітектуру.**
   Архітектура, де функціональність системи розбита на незалежні сервіси, що спілкуються через стандартизовані інтерфейси.

3. **Якими принципами керується SOA?**
   Слабке зв’язування, повторне використання сервісів, чіткі контракти, незалежність платформи, автономність, композиційність.

4. **Як між собою взаємодіють сервіси в SOA?**
   Через стандартизовані протоколи й API (HTTP, SOAP, REST, JMS) за допомогою обміну повідомленнями.

5. **Як розробники взнають про існуючі сервіси і як робити до них запити?**
   Через сервісний реєстр/каталог (наприклад, WSDL або API документацію), де описані інтерфейси та адреси виклику.

6. **Переваги та недоліки клієнт-серверної моделі?**
   Переваги: централізація даних, легке адміністрування, масштабованість.
   Недоліки: залежність від сервера, можливі вузькі місця, складність масштабування серверної частини.

7. **Переваги та недоліки однорангової моделі?**
   Переваги: відсутність централізованої точки збою, висока масштабованість, рівноправність вузлів.
   Недоліки: складність координації, нестабільність продуктивності, проблеми безпеки.

8. **Що таке мікросервісна архітектура?**
   Підхід, коли система складається з маленьких незалежних сервісів, кожен з яких виконує одну бізнес-функцію та розгортається окремо.

9. **Які протоколи використовуються для обміну даними в мікросервісній архітектурі?**
   HTTP/REST, gRPC, AMQP, WebSockets, Kafka, MQTT.

10. **Чи можна назвати підхід SOA, коли між контролерами та DAO створюється шар сервісів?**
    Ні, це просто багатошарова архітектура; справжня SOA передбачає розподілені сервіси з незалежним розгортанням і стандартними контрактами.