Specifica Tecnica

Gruppo SWEet BIT — Progetto SWEDesigner

Informazioni sul documento

Versione	1.0.0
Redazione	Santimaria Davide
	Massignan Fabio
Verifica	Massignan Fabio
	Bodian Malick
Approvazione	Pilò Salvatore
Uso	Esterno
Distribuzione	Prof. Tullio Vardanega
	Prof. Riccardo Cardin
	Gruppo SWEet BIT
	Zucchetti S.p.A.

Descrizione

Questo documento descrive la specifica tecnica e l'architettura del prodotto sviluppato dal gruppo SWEet BIT per la realizzazione del progetto SWEDesigner.

Versioni del documento

Versione	Data	Persone coinvolte	Descrizione
1.y.z	2017/??/??	Pilò Salvatore	Approvazione Documento
1.y.z	2017/??/??	Massignan Fabio	Verifica Documento
1.0.3	2017/05/02	NOME	Stesura sezione Descrizione architettura
1.0.2	2017/05/02	NOME	Stesura sezione Tecnologie utilizzate
1.0.1	2017/05/02	NOME	Stesura sezione Introduzione
1.0.0	2017/05/02	Santimaria Davide	Creazione struttura documento

Indice

1 Introduzione

1.1 Scopo del documento

Questo documento ha come scopo quello di definire la *progettazione ad alto livello*$_G$ per il prodotto. Verrà presentata la strttura generale secondo la quale saranno organizzate le varie componenti software e i *Design Pattern*$_G$ utilizzati nella creazione del prodotto SWEDesigner. Verrà dettagliato il tracciamento tra le componenti software individuate ed i requisiti.

1.2 Scopo del prodotto

Lo scopo del progetto è la realizzazone di una *Web App*$_G$ che fornisca all'*Utente*$_G$ un *UML*$_G$ *Designer*$_G$ con il quale riuscire a disegnare correttamente *Diagrammi*$_G$ delle *Classi*$_G$ e descrivere il comportamento dei *Metodi*$_G$ interni alle stesse attraverso l'utilizzo di *Diagrammi*$_G$ delle attività. La *Web App*$_G$ permetterà all'*Utente*$_G$ di generare *Codice*$_G$ *Java*$_G$ dall'insieme dei *diagrammi classi*$_G$ e dei rispettivi *metodi*$_G$.

1.3 Glossario

Con lo scopo di evitare ambiguità di linguaggio e di massimizzare la comprensione dei documenti, il gruppo ha steso un documento interno che è il *Glossario v*1.2.0. In esso saranno definiti, in modo chiaro e conciso i termini che possono causare ambiguità o incomprensione del testo.

1.4 Riferimenti

1.4.1 Normativi

- **Capitolato d'Appalto C6: SWEDesigner**
  http://www.math.unipd.it/~tullio/IS-1/2015/Progetto/C6p.pdf;
- **Norme di Progetto:** *Norme di Progetto v1.2.0.*
- **Analisi dei Requisiti:** *Analisi dei Requisiti v1.2.0.*

1.4.2 Informativi

- **Slide dell'insegnamento Ingegneria del Software modulo A:** http://www.math.unipd.it/~tullio/IS-1/2016/.

  - **Slides del corso di Ingegneria del Software mod. A: *Diagrammi delle classi*$_G$**: http://www.math.unipd.it/~tullio/IS-1/2015/Dispense/E03.pdf;
  - **Slides del corso di Ingegneria del Software mod. A: Diagrammi dei package**: http://www.math.unipd.it/~tullio/IS-1/2015/Dispense/E04.pdf;
  - **Slides del corso di Ingegneria del Software mod. A: Diagrammi di sequenza**: http://www.math.unipd.it/~tullio/IS-1/2015/Dispense/E05.pdf;
  - **Slides del corso di Ingegneria del Software mod. A: Diagrammi di attività**: http://www.math.unipd.it/~tullio/IS-1/2015/Dispense/E06.pdf;
  - **Slides del corso di Ingegneria del Software mod. A: *Design pattern*$_G$ strutturali: Decorator, Proxy, Facade, Adapter**:http://www.math.unipd.it/~tullio/IS-1/2015/Dispense/E07.pdf;
  - **Slides del corso di Ingegneria del Software mod. A: *Design pattern*$_G$ creazionali: Singleton, Builder, Abstract Factory**: http://www.math.unipd.it/~tullio/IS-1/2015/Dispense/E08.pdf;
  - **Slides del corso di Ingegneria del Software mod. A: *Design pattern*$_G$ comportamentali: Observer, Template Method, Command, Strategy, Iterator**: http://www.math.unipd.it/~tullio/IS-1/2015/Dispense/E09.pdf;

- **Design Patterns - E. Gamma, R. Helm, R. Johnson, J. Vlissides (Pearson Education, Addison-Wesley, 1995;**;
- ***Node.js*$_G$**: https://nodejs.org/dist/latest-v5.x/docs/api/;
- **MongoDB**: https://docs.mongodb.org/manual/;
- **HTML5**: http://www.w3schools.com/html/html5_intro.asp;
- **CSS3**: http://www.w3schools.com/css/css3_intro.asp;
- **ExpressJS**: http://expressjs.com/en/4x/api.html.

2 Tecnologie utilizzate

L'architettura è stata progettata utilizzando lo stack di ***MEAN***$_G$ (http://mean.io/),il quale comprende 4 tecnologie, alcune delle quali espressamente richieste nel *capitolato*$_G$ d'appalto. Vengono di seguito elencate e descritte le principali tecnologie impiegate comprese in ***MEAN***$_G$ e le motivazioni del loro utilizzo:

- **Node.js**: piattaforma per il *back-end*$_G$;
- **Express.js**: *framework*$_G$ per la realizzazione dell'applicazione web in *Node.js*$_G$;
- **MongoDB**: *database*$_G$ di tipo *NoSQL*$_G$ per la parte di recupero e salvataggio dei dati;
- **Mongoose**: *libreria*$_G$ per interfacciarsi con il driver di **MongoDB**;
- **Angular.js**: *framework*$_G$ *JavaScript*$_G$ la realizzazione del *front-end*$_G$.

2.1 Server

2.1.1 Node.js

Node.js è una *piattaforma*$_G$ software costruita sul motore *JavaScript*$_G$ di *Chrome*$_G$ che permette di realizzare facilmente applicazioni di rete scalabili e veloci. *Node.js*$_G$ utilizza *JavaScript*$_G$ come linguaggio di programmazione, e grazie al suo modello *event-driven*$_G$ con chiamate di input/output non bloccanti risulta essere leggero e e ciente.

2.1.1.1 Vantaggi

- **Approccio asincrono**: *Node.js*$_G$ permette di accedere alle risorse del sistema operativo in modalità *event-driven*$_G$ e non sfruttando il classico modello basato su processi concorrenti utilizzato dai classici web *server*$_G$. Ciò garantisce una maggiore efficienza in termini di prestazioni, poiché durante le attese il runtime può gestire qualcos'altro in maniera asincrona;
- **Architettura modulare**: Lavorando con *Node.js*$_G$ è molto facile organizzare il lavoro in librerie, importare i *moduli*$_G$ e combinarli fra loro.Questo è reso molto comodo attraverso il *node package manager*$_G$ (**npm**) attraverso il quale lo sviluppatore può contribuire e accedere ai *package*$_G$ messi a disposizione dalla community.

2.1.1.2 Svantaggi

- **Supporto incompleto alle feature di ES6**: Molte delle feature di ES6 non sono supportate in Node nella versione 4.4 scelta come versione di riferimento per lo sviluppo del progetto.

2.2 Client

2.2.1 Express.js

Express.js è un *framework*$_G$ minimale per creare *Web App*$_G$ con *Node.js*$_G$. Richiede *moduli*$_G$ Node di terze parti per applicazioni che prevedono l'interazione con le *database*$_G$. È stato utilizzato il *framework*$_G$ *Express.js*$_G$ per supportare lo sviluppo dell'application *server*$_G$ grazie alle utili e robuste caratteristiche da esso offerte, le quali sono pensate per non oscurare le funzionalità fornite da *Node.js*$_G$ aprendo così le porte all'utilizzo di moduli per *Node.js*$_G$ atti a supportare specifiche funzionalità.

2.2.1.1 Vantaggi

- **Minimale**: si basa su *Node.js*$_G$ e permette di estenderlo a seconda dei bisogni dell'applicazione;
- **Documentazione**: esaustiva e completa;
- **Apprendimento**: facile da imparare.

2.2.1.2 Svantaggi

- **Integrazione**: richiede di integrare *moduli*$_G$ diversi per comporre l'applicazione finale. Altri *framework*$_G$ permettono di definire *API*$_G$ (Application Programming Interface) *REST*$_G$ (REpresentational State Transfer) in modo semplice, ma vincolano maggiormente nelle scelte progettuali.

2.2.2 MongoDB

MongoDB$_G$ è un *database*$_G$ *NoSQL*$_G$ *open source*$_G$ scalabile e altamente performante di tipo document-oriented, in cui i dati sono archiviati sotto forma di documenti in stile *JSON*$_G$ con schemi dinamici, secondo una struttura semplice e potente.

2.2.2.1 Vantaggi

- **Alte performance**: non ci sono join che possono rallentare le operazioni di lettura o scrittura. L'indicizzazione include gli indici di chiave anche sui documenti innestati e sugli array, permettendo una rapida interrogazione al *database*$_G$;
- **Affidabilità**: alto meccanismo di replicazione su server;
- **Schemaless**: non esiste nessuno *schema*$_G$, è più flessibile e può essere facilmente traspostoin un modello ad oggetti;

- Permette di definire query complesse utilizzando un linguaggio che non è SQL_G;
- Permette di processare parallelamente i dati ($Map\text{-}Reduce_G$);
- Tipi di dato più flessibili.

2.2.2.2 Svantaggi

- **Flessibilità**: per i tipi di dato. Sebbene questo possa essere visto come vantaggio, è opinione del team che un'eccessiva flessibilità possa portare più problemi che benefici: allo scopo di aggiungere rigidità è stato infatti scelto, come verrà descritto in seguito, Mongoose, che introduce una costruzione a schemi per le collections di $MongoDB_G$ e quindi vincola i documenti inseriti ad avere una struttura uniforme;
- **Nessun supporto per le transazioni**: sono supportate alcune operazioni atomiche, ma a livello di documento;
- **Nessun *join*$_G$**: va simulato via codice attraverso query multiple;
- **Problemi di concorrenza**: per le operazioni di scrittura viene creato un lock sull'intero database. Questo lock blocca anche le operazioni di lettura.

2.2.3 Mongoose

Mongoose è una $libreria_G$ per interfacciarsi a $MongoDB_G$ che permette di definire degli schemi per modellare i dati del $database_G$, imponendo una certa struttura per la creazione di nuovi Document . Inoltre fornisce molti strumenti utili per la validazione dei dati, per la definizione di query e per il cast dei tipi predefiniti. Per interfacciare l'application $server_G$ con $MongoDB_G$ sono disponibili diversi progetti $open\ source_G$. Per questo progetto è stato scelto di utilizzare $Mongoose.js_G$, attualmente il più di uso.

2.2.3.1 Vantaggi

- **Diffusione**: è la libreria più diffusa per interfacciarsi con $MongoDB_G$;
- **Funzionalità aggiuntive**: permette di definire strumenti per la validazione dei dati e per il cast dei tipi;
- **Permette di eseguire dei *join*$_G$ tra collections**: Sebbene non sia previsto da $MongoDB_G$, $mongoose_G$ prevede la funzione populate per imitare la funzione di $join_G$ in modo completamente trasparente per l'utilizzatore;
- **Rapido ed intuitivo**: La strutturazione dei dati con questa libreria è rapida ed inutitiva, ciò dovuto anche dalla sintassi dichiarativa della libreria stessa.

2.2.3.2 Svantaggi

- **Schema-based**: è basato sulla creazione di una forte schematizzazione per i documenti, e questo limita l'estrema flessibilità di *MongoDB*$_G$.

2.2.4 mxGrafh

mxGraph è una libreria *JavaScript*$_G$ di *diagrammi*$_G$ che consente di creare rapidamente applicazioni di grafici interattive e grafici che vengono eseguiti in modo nativo su tutti i browser principali.

2.2.4.1 Vantaggi

- **Non sono necessari altri plug-in**: Ciò elimina i plug-in di dipendenza dai fornitori;
- ***Open-source***$_G$:Le tecnologie coinvolte sono libere e ci sono molte implementazioni aperte, nessun fornitore può rimuovere un prodotto o una tecnologia che lascia in pratica la tua applicazione inoperabile;
- **Le tecnologie sono standardizzate**: L'applicazione è distribuibile al numero massimo di utenti del browser senza bisogno di ulteriori configurazioni o installazione sul computer *client*$_G$. Gli ambienti aziendali di grandi dimensioni spesso non amano consentire agli individui di installare plug-in del *browser*$_G$ e non amano cambiare la build standard creata su tutte le macchine.

2.2.4.2 Svantaggi

- **Aumento rapido di celle**: Poiché il numero di celle visibili sullo schermo degli utenti aumenta di centinaia, la valutazione rallenta oltre i limiti accettabili sulla maggior parte dei *browser*$_G$. Nella teoria della gestione delle informazioni, visualizzare diverse centinaia di celle è generalmente sbagliato, in quanto l'utente non può interpretare i dati.

3 Descrizione architettura

3.1 Metodo e formalismo di specifica

Le scelte architetturali per lo sviluppo di SWEDesigner sono state fortemente influenzate dallo stack tecnologico utilizzato.

Nell'esposizione dell'architettura dell'applicazione si procederà con un approccio *top-down*$_G$, descrivendo l'architettura iniziando dal generale ed andando al particolare; si è partiti suddividendo il sistema in *front-end*$_G$ e *back-end*$_G$, definendo l'interfaccia di comunicazione, scegliendo di seguire in ciascuno l'organizzazione suggeritaci dai *framework*$_G$.

La descrizione dell'architettura di SWEDesigner è suddivisa in quattro sezioni:

- §3.2: illustra gli aspetti generali dell'architettura del software;
- §3.3: descrive il protocollo che lega le due interfacce tra *Client*$_G$ e *Server*$_G$; che descrive l'architettura del front end dell'applicazione;
- §3.4: descrive l'architettura del *back-end*$_G$ dell'applicazione;
- §3.5: descrive l'architettura del *front-end*$_G$ dell'applicazione.

Per descrivere in maniera formale l'architettura verranno impiegati lo standard *UML*$_G$ 2.0 per i diagrammi dei *package*$_G$ e delle classi e lo standard *UML*$_G$ 2.4 per i *diagrammi di attività*$_G$ e sequenza.

I diagrammi delle *classi*$_G$ che permettono di mostrare l'architettura generale del sistema vengono affiancati anche dai diagrammi di sequenza e attività, che permettono di definire le interazioni tra le componenti, senza preoccuparsi della loro classificazione. In questo modo è possibile esprimere alcuni meccanismi tipici di un'applicazione *REST*$_G$-like, come il modo in cui agiscono i *middleware*$_G$.

3.2 Architettura generale

L'architettura del progetto si divide in una componente *Client*$_G$, rappresentata da un'applicazione *front-end*$_G$ accessibile da un browser, e in una componente Web*Server*$_G$, nella quale risiede il *back-end*$_G$ che gestisce le richieste di generazione del *codice*$_G$.
L'architettura generale di SWEDesigner si divide in 3 macrocomponenti:

- ***Client***$_G$;
- ***Server***$_G$ ***REST***$_G$;
- ***Database***$_G$.

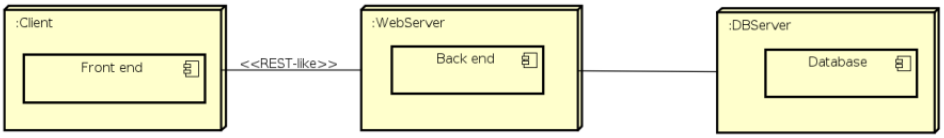

Figura 1: Diagramma di *deployment*$_G$ per l'architettura

L'architettura proposta segue il *Design Pattern*$_G$ MVC. In particolare i ruoli di Model e Controller verranno implementati a livello di *server*$_G$, mentre il ruolo di View viene affidato al *front-end*$_G$. L'interfaccia tra le due componenti verrà gestita grazie ad un set di *API*$_G$ disposto dal *server*$_G$ *REST*$_G$; il *Database*$_G$ serve per garantire la persistenza del programma generato: ogni *utente*$_G$ autenticato può salvare i propri progetti e mantenere i diagrammi creati.
Le tre macrocomponenti verranno descritte in dettaglio in seguito su questo documento.

3.3 Interfaccia REST-like

Per l'interfaccia della componente *back-end*$_G$ si è scelto di utilizzare uno stile basato REST. All'interno di un'unica sessione utente, a partire dall'operazione di login fino a quella di logout, l'interfaccia con cui si accede agli elementi delle collection può considerarsi e ettivamente REST.
I motivi che hanno spinto alla scelta di RESTsono:

- Semplicità di utilizzo;
- Facile integrazione con i *framework*$_G$ esistenti;
- Indipendenza dal linguaggio di programmazione utilizzato.

REST utilizza il concetto di risorsa, ovvero un aggregato di dati con un nome (URIG) e una rappresentazione, su cui è possibile invocare le operazioni CRUD tramite la seguente corrispondenza:

3.4 Architettura del Server

3.5 Architettura del Client

4 Back-end

4.1 Interfaccia REST

4.2 Descrizione packages e classi

4.2.1 Back-end

4.2.1.1 Informazioni sul package

4.2.2 Back-end::Lib

4.2.2.1 Informazioni sul package

4.2.3 Back-end::Lib::AuthModel

4.2.3.1 Informazioni sul package

4.2.3.2 Classi

4.2.4 Back-end::Lib::Whatever

4.2.4.1 Informazioni sul package

4.2.4.2 Classi

4.3 Scenari

4.3.1 Gestione generale delle richieste

4.3.2 Fallimento vincolo "utente autenticato"

4.3.3 Fallimento vincolo "utente non autenticato"

4.3.4 Richiesta POST /login

4.3.5 Richiesta DELETE /logout

4.4 Descrizione librerie aggiuntive

5 Front-end

5.1 Descrizione packages e classi

5.1.1 Front-end

5.1.1.1 Informazioni sul package

5.1.2 Front-end::Controllers

5.1.2.1 Informazioni sul package

5.1.2.2 Classi

5.1.3 Front-end::Services

5.1.3.1 Informazioni sul package

5.1.3.2 Classi

5.1.4 Front-end::Model

5.1.4.1 Informazioni sul package

5.1.4.2 Classi

6 Diagrammi delle attività

6.1 Applicazione SWEDwsigner

6.1.1 Attività principali

6.1.2 Registrazione

6.1.3 Recupero password

6.1.4 Login

6.1.5 Modifica profilo

6.1.6 Altro

7 Stime di fattibilità e di bisogno e di risorse

8 Design pattern

8.1 Design Pattern Architetturali

8.1.1 MVVM

8.1.2 Dependency Injection

8.2 Design Pattern Creazionali

8.2.1 Factory ad esempio

8.3 Design Pattern Strutturali

8.3.1 Decorator

8.3.2 Facede

8.4 Design Pattern Comportamentali

8.4.1 Observer

8.4.2 Command

9 Tracciamento

9.1 Tracciamento componenti - requisiti

9.2 Tracciamento requisiti - componenti

10 Appendici

A Descrizione Design Pattern

A.1 Design Pattern Architetturali

A.1.1 MVVM

A.1.2 Dependency Injection

A.2 Design Pattern Creazionali

A.2.1 Factory ad esempio

A.3 Design Pattern Strutturali

A.3.1 Decorator

A.3.2 Facede

A.4 Design Pattern Comportamentali

A.4.1 Observer

A.4.2 Command