PROTOCOLLO HTTP

HTTP è un protocollo di trasporto di contenuti generici alla base del web e di tutte le applicazioni costruite su tale piattaforma. Si appoggia ad un protocollo di trasporto esistente che nella maggior parte dei casi è il TCP.
Gestisce uno scambio elementare: il client invia una richiesta e il server produce una risposta: ognuna di queste richieste sono indipendenti così come è indipendente l'utilizzo di tale protocollo rispetto al contenuto.

Sia la richiesta che la risposta sono formate da un'intestazione e da un corpo (il cui contenuto è definito nell'intestazione).
Tutte le transazioni HTTP riguardano operazioni su risorse informative specificate attraverso una URL, una stringa univoca costituita da più sottoparti (schema://hostname[:port]/path):

- schema indica il protocollo tramite cui agire
- hostname[:port] è l'indirizzo in cui il server si trova in attesa di connessioni
- path indica la posizione relativa dell'oggetto all'interno del server

La prima riga di una richiesta contiene:

- L'azione da svolgere: GET (richiede l'invio della risorsa indicata dalla URL), POST (invia un insieme di informazioni alla risorsa, tipicamente un programma, indicata dalla URL e richiede il documento risultante), PUT (richiede l'aggiornamento di una risorsa sul server identificata dalla URL), PATCH (richiede l'aggiornamento parziale di una risorsa), DELETE (richiede la distruzione della risorsa indicata)
- La URL relativa dell'oggetto su cui operare
- La versione del protocollo usata dal client

Un'azione è idempotente se l'effetto sul server di più richieste identiche è lo stesso di quello di una sola richiesta ed è sicura se non genera cambiamenti nello stato interno del server.

Le risposte rappresentano il risultato della richiesta tramite un codice ed una descrizione. Il sostanziale mutare nel tempo dei tipici contenuti trasferiti ha reso il meccanisco di trasporto poco efficiente (la latenza è un fattore critico): con HTTP/2, su una singola connessione TCP vengono multiplate più transazioni, ognuna delle quali è incapsulata in un pacchetto binario che la identifica. Più richieste possono essere emesse in cascata e le risposte non devono seguire l'ordine delle richieste: il pacchetto permette di ricostruire a quale richiesta faceva riferimento.

APPLICAZIONI HTTP

Lo standard prevede quattro possibili ruoli per le applicazioni che utilizzano HTTP:

- User-agent: applicazione che origina le richieste HTTP
- (Origin) Server: applicazione che ospita le risorse trasferibili. Accetta quindi richieste e genera le corrispondenti risorse
- Proxy: intermediario scelto dallo user-agent per inoltrare le richieste al server. Può filtrare le richieste, tradurre indirizzi e servirsi di una memoria temporanea per aumentare le prestazioni
- Gateway: intermediario trasparente allo user-agent che inoltra le richieste all'origin effettivo

Esistono due tipologie di proxy: quelli trasparenti dove la risposta viene inoltrata al client senza modifiche (è comunque capace di usare una cache e possiede un sistema di filtraggio) e quelli non trasparenti, che inoltrano tutte le richieste ma hanno la possibilità di modificare le risposte.
Basic Authentication: ad ogni gruppo di risorse tra loro correlate è associato un contesto di sicurezza chiamato realm. Per ogni realm è definita una lista di utenti autorizzati, con le relative password: quando il server manda una 401-Unauthorized, specifica il realm-name ed è compito del client formulare una nuova richiesta (dopo aver chiesto all'utente user e password) con le informazioni necessarie codificate in base64 (che però è una codifica reversibile quindi OCCHIO).

Digest authentication: invece di inviare la password, è possibile inviare un dato derivato in modo irreversibile dalla password. Client e Server eseguono la derivazione in modo indipendente, cosicchè se le due coincidono, il client ha dimostrato di conoscere la password senza inviarla in chiaro; occorre difendersi da tentativi di proporre la stessa derivazione da parte di un'eventuale attaccante.

Poichè HTTP tratta tutte le richieste come tra loro indipendenti, occorre introdurre a livello applicativo un meccanismo per supportare questa modalità di lavoro. Nasce così il concetto di sessione: un insieme di richieste, proveniente dallo stessso browser e dirette allo stesso server, confinate in un dato lasso di tempo, volte ad interagire con risorse tra loro correlate. E' possibile sia creare sessioni nominali che sessioni anonime.
Il web server associa una richiesta proveniente da un dato client ad un identificatore univoco (SessionID): tale identificatore viene comunicato al client per le successive richieste. Dopo un certo periodo di inattività dell'utente, la sessione scade e ne viene perso lo stato; l'ID può essere sia generato sequenzialmente a partire da un valore dato oppure essere generato casualmente. In ogni caso, tale ID viene trasferito al client in diversi modi: passaggio esplicito nel contenuto del documento, utilizzando i cookies, riscrivendo i link o coordinando le sessioni lato client.

ANNOTAZIONI JAVA

Le annotazioni permettono di inserire dei metadati che descrivono le caratteristiche di una classe. Esse possono essere usate per informare il compilatore (rilevare errori o evitare i warning), generare informazioni durante la compilazione di un pacchetto software o condizionare l'esecuzione di un programma (è possibile esaminare una classe durante l'esecuzione per verificare la presenza di annotazioni e comportarsi di conseguenza).

```
public @interface RequestForEnhancement{
        int id();
        String synopsis();
        String date();
}
```

Il simbolo @ precede la keyword interface e i metodi definiscono le proprietà dell'annotazione. Una volta definito il tipo di annotazione, questo può essere usato per annotare delle dichiarazioni. Le dichiarazioni dei metodi non devono avere parametri e non devono prevedere clausole di tipo throws, infine i tipi di ritorno devono essere tipi primitivi, String, Class, enum o array di questi.
Le meta-annotazioni servono per annotare le dichiarazioni delle annotazioni (annotazioni di annotazioni), esempi sono: @Target, @Retention, @Documented, @Inherited. Usando JPA è possibile anche configurare il comportamento delle proprie entità usando le annotazioni (come @Entity, @OneToOne, @PrimaryKey o @Column).
Con la nascita delle annotazioni è nato anche il progetto Lombok, che è un'estensione del compilatore in grado di convertire annotazioni presenti nelle classi in codice sorgente Java e quindi permette di trasformare un oggetto annotato con @Data aggiungendo getter/setter/costruttore in automatico. Le classi annotate con @Data, inoltre, hanno tutti i campi non esplicitamente final mutabili, mentre annotando con @Value tutti i campi vengono resi privati e immutabili. In conclusione, le annotazioni Java rappresentano una standardizzazione dell'approccio di annotazione del codice: non riguardano direttamente la semantica del programma ma possono semplificare notevolmente la quantità di codice che è necessario scrivere a mano senza compromettere le funzionalità.

MAVEN

Maven è uno strumento a supporto del processo di creazione e manutenzione del software. E' una versione evoluta di make, con supporto all'importazione delle dipendenze, alla creazione di moduli differenti ed all'utilizzo di plugin per la compilazione e la messa in campo. Supporta le diverse fasi del ciclo di vita di un prodotto software, e le singole applicazioni sono eseguite da plugin.
E' necessario un file POM (Project Object Model), un file XML che descrive la struttura di un progetto; il progetto e ciascun artefatto da esso utilizzato vengono identificati in modo univoco tramite la notazione GAV(groupId : artifactId : version):

- groupId: identificatore arbitrario del progetto di solito derivato dal package Java
- artifactId: nome arbitrario del progetto

- version: versione del progetto

Un file POM può ereditare la propria configurazione e in tal modo facilita la gestione di progetti composti da più sotto-progetti.
Un progetto Maven può essere costituito da un insieme di moduli ognuno dei quali è contenuto in una sotto-cartella ed ha un proprio file POM; nella cartella del progetto è presente il file POM complessivo: target è la cartella di lavoro di default e src contiene tutti i file sorgente.

Per default, la creazione di un artefatto passa attraverso le seguenti fasi principali: generazione dei sorgenti e delle risorse (generate*, qui vengono eseguiti eventuali pre-processor del codice sorgente), compilazione (compile), esecuzione dei test sui moduli (test), impacchettamento (package), test di integrazione (integration-testing), installazione (install) e messa in campo (depoly). Separatamente viene considerata la fase di pulizia generale (clean, che elimina tutti i file intermedi e di uscita).
Nella maggior parte dei casi, un progetto reale si basa su un insieme di altri progetti/librerie. Per ogni elemento di tipo <dependency> occorre indicare GroupId, ArtifactId, Version, Scope; quest'ultimo <scope> definisce in quale fase del ciclo di vita del progetto la dipendenza dovrà essere considerata: compile (indica che la risorsa è necessaria per l'intero ciclo di vita), provided (indica che la risorsa è necessaria durante la fase di compilazione, ma non dovrà essere inclusa nell'artefatto finale in quanto messa a disposizione dall'ambiente di esecuzione), runtime (indica una risorsa da includere solo in fase di esecuzione) e test (indica una risorsa necessaria esclusivamente in fase di test).

Per ogni risorsa indicata come dipendenza, il corrispondete modulo software viene scaricato dal repository ed archiviato nella cache locale (viene scaricato inoltre anche il file POM corrispondente). Se, dall'analisi del POM, un modulo scaricato dichiaradi dipendere da altri moduli, questi vengono scaricati transitivamente (ma solo se sono "compile" e "runtime").
Tutte le operazioni svolte da Maven avvengono grazie all'attivazione di appositi plugin, ossia classi Java che estendono una classe astratta e implementano il metodo void execute(); un'apposita sezione del file POM (<build>) elenca l'insieme di plugin che saranno utilizzati nella gestione del progetto.

APPLICAZIONI JAVAEE

Le applicazione web dinamiche implementano una data logica lato server fornendo un'interfaccia utente basata su pagine html e contenuti multimediali: il livello UI viene eseguito sul client e mostra l'interfaccia utente, il livello controllo viene eseguito sul server e si occupa di implementare la logica e il livello dati viene eseguito su una base dati e si occupa di memorizzare i dati applicativi.Contrariamente alle applicazioni stand alone, quelle web sono eseguite in modo reattivo (tutte le volte che ricevono una richiesta producono una risposta specifica per la sessione in corso): solitamente, l'applicazione reagisce alla prima richiesta mostrando una "home page". Ciascuna richiesta può contenere parametri aggiuntivi (tramite form ecc..) e il livello di controllo deve: verificare la validità dei parametri ricevuti, memorizzare temporaneamente i parametri per poterli utilizzare e generare una risposta appropriata, permettendo all'utente di svolegere il proprio compito. La pagina inviata può poi essere personalizzata in base alle interazioni precedenti.

Nella navigazione pagina per pagina, a fronte di ogni richiesta viene generata una risposta che contiene una pagina HTML completa, mentre in quella progressiva a pagina singola solo la prima richiesta contiene la struttura della pagina, e le richieste successive sono effettuate con il paradigma AJAX e prevedono il trasferimento solo del frammento di struttura che deve essere modificato o di una descrizione sintetica dello stesso.
La navigazione a pagina singola è basata sull'utilizzo di codice JavaScript lato client per richiedere il frammento da aggiornare e si utilizzano le funzioni per la manipolazione del DOM per inserirle nella pagina; l'uso di librerie quali Vue, React o Angular semplifica la gestione del processo. I limiti di tale navigazione si incontrano quando ci si interfaccia con i motori di ricerca, con il tasto "indietro" dei browser e per i tempi di caricamento.

Le applicazioni JavaEE sono un insieme modulare di classi Java, messe in campo all'interno di un programma contenitore che ne gestisce il ciclo di vita e l'esecuzione: esse sono costituite da un insieme di componenti responsabili di gestire l'elaborazione di richieste HTTP e costruire le corrispondenti risposte, delegando in parte o in toto l'implementazione della logica applicativa ad uno strato di servizio che, a sua volta, può appoggiarsi ad uno strato di memorizzazione persistente. Esse sono indipendenti dallo specifico contenitore, ossia applicazioni in esecuzione all'interno di un calcolatore server. Monitorano le connessioni, applicano ad ogni richiesta un insieme di funzioni

base, determinano l'applicazione cui è destinata la richiesta e la classe al suo interno responsabile di elaborarla, instanziandola se necessario e offrono un insieme di servizi standardizzati ai singoli componenti, semplificandone l'implementazione. Utilizzano un template standard per descrivere le applicazioni web ed hanno un file XML di configurazione da cui il contenitore deduce tutti i dettagli applicativi (politiche di gestione, parametri di configurazione, corrispondenza fra URL e nomi delle classi ad esempio).

Le applicazioni JavaEE sono normalmente distribuite sotto forma di file ".war", al cui interno sono presenti sia i contenuti che la configurazione dell'applicazione e il cui nome rappresenta di default la URL di base in cui verrà pubblicata. Ci sono due sotto-cartelle obbligatorie (META-INF/MANIFEST.MF e WEB-INF/web.xml) e per installarla basta semplicemente copiare il file .war nell'apposita cartella del contenitore. Ci sono poi le componenti elementari di un'applicazione JavaEE (Servlet, filtri, Listener, pagine JSP).

SERVLET API

Un Servlet è una classe Java responsabile di gestire le richieste ricevute da un contenitore, modellando il paradigma richiesta/elaborazione/risposta tipico delle applicazioni lato server. Il contenitore ne gestisce completamente il ciclo di vita. Le sue API sono un insieme di interfacce e classi che definiscono, a basso livello, l'interazione tra codice applicativo lato server e contenitore. javax.servlet contiene le classi e le interfacce che definiscono il comportamento generico di un servlet, javax.servlet.http contiene le classi e le interfacce che modellano il comportamento di richieste/risposte trasportate attraverso il protocollo HTTP. ServletRequest rappresenta la richiesta del client e ServletResponse rappresenta la risposta del servlet al client (estese da HTTPServletRequest e HTTPServletResponse).

Il contenitore crea un'unica istanza del servlet e ne invoca il metodo init(), e finchè esso non ritorna il contenitore non invoca nessun altro metodo. Ad ogni richiesta, il contenitore invoca il metodo service() utilizzando un thread differente e infine, a proprio piacimento, il contenitore decide di invocare il metodo destroy() per rimuovere il servlet. L'annotazione WebServlet permette di indicare la configurazione: il valore di default indica l'elenco di URL che vengono gestite dall'istanza del servlet, urlPatterns è equivalente al valore di default, initParams indica eventuali parametri di inizializzazione, loadOnStartup indica la politica di instanza, asyncSupported indica se la richiesta può essere gestita in modalità asincrona.

Metodi di HTTPServletRequest: getParameter(String name), ServletInputStream getInputStream(), BufferedReader getReader(), String getMethod(), Enumeration getParameterNames(), String getHeader(String headerName), void setAttribute(String name, Object value), Object getAttribute(String name)

Metodi di HTTPServletResponse: void setContentType(String contentType), void setStatusCode(int statusCode), void addCookie(Cookie c), void addHeader(String name, String value).

Metodi di HTTPSession: void setAttribute(String name, Object value), Object getAttribute(String name), void removeAttribute(String name), String getId(), boolean isNew(), void invalidate()

Il ServletContext è un'interfaccia che definisce le interazioni tra servlet e suo contenitore. Nel caso di contenitori basati su una singola virtual machine, ogni applicazione web ha un unico contesto, mentre nel caso di più VM ci sono tante istanze di questa interfaccia quante sono le macchine virtuali attive.
In ogni caso, permette di creare/aggiungere/rimuovere servlet/filtri/listener al contenitore ed offre un meccanismo per risalire al tipo di contenuto di un file presente nel file system locale; inoltre, è in grado di fornire informazioni sul contenitore, sulle API supportate e sulle politiche in essere per la gestione dell'applicazione.

Quali sono i vantaggi di un Servlet? Di sicuro sono veloci, efficienti, scalabili, flessibili e persistenti; in più, il contenitore garantisce la separazione fisica tra applicazioni. Il problema è che è complicato modificare il codice HTML di risposta, la presentazione e la logica sono fortemente accoppiate e l'architettura rischia di diventare complessa se si prendono in considerazione le caratteristiche più avanzate.

FILTRI

Un filtro è un componente java che trasforma le richieste inviate ad un servlet o le risposte da esso generate. Ciascun filtro riceve le richieste destinate ad un (o un insieme di) URL e decide, in base alla propria logica, se inoltrare la

richiesta al destinatario, modificare uno o più parametri o reindirizzare la richiesta verso un indirizzo differente. Lo stesso funzionamento vale anche per la risposta, e ci possono essere più filtri collegati in cascata, il tutto gestito dal file web.xml.
Come nel caso dei Servlet, anche per i filtri è possibile indicare il set di URL su cui devono essere applicati in due modi alternativi: tramite l'annotazione @WebFilter o inserendo delle sezioni nel file web.xml. Parametri:

- urlPatterns – indica la lista delle espressioni regolari che descrivono l'insieme delle URL su cui il filtro si applica
- servletNames – indica l'insieme dei servlet su cui deve essere applicato il filtro
- initParams – i parametri di configurazione del filtro
- filterName – il nome simbolico associato al filtro
- dispatcherType – indica a quale fase della gestione della richiesta debba essere applicato il filtro
- asynchSupported – indica se il filtro può operare in modalità asincrona

Ci sono due sezioni nel file web.xml: "filter" descrive un filtro mentre "filter-mapping" esprime la corrispondenza tra URL e filtri da utilizzare. Nel caso in cui ad una data URL corrispondano due o più filtri, questi sono applicati nell'ordine in cui le rispettive sezioni filter-mapping compaiono nel file web.xml

JSP - JAVA SERVER PAGES

Documenti di testo che descrivono come elaborare una richiesta per produrre una risposta in base ai parametri ricevuti. JSP è un modello di programmazione basata sul linguaggio Java e usato per creare dinamicamente contenuti web. Le pagine JSP contengono testo, frammenti di codice e direttive per l'ambiente di esecuzione e vengono create e gestite nel contenitore. Esse generano contenuto dinamico che viene visualizzato in un browser generico (nonostante il client non veda mai la pagina JSP iniziale, ma solo i risultati della sua esecuzione) grazie ai frammenti di codice Java, per cui semplici oggetti (JavaBeans) possono essere utilizzati per incapsulare data storage o altre operazioni di tipo business.
Che vantaggi porta l'utilizzo di JSP? Si può modificare il contenuto delle pagine web senza cambiare il codice HTML di layout e vi è quindi una separazione tra i contenuti dinamici e quelli statici; inoltre, i JavaBeans e i tag personalizzati permettono il riutilizzo di codice e l'utilizzo di semantiche dichiarative. L'unico problema è che il programmatore deve occuparsi di progettare ed implementare una macchina a stati opportuna per gestire l'interazione con l'utente.

1. Il contenitore riceve la richiesta della pagina JSP (direttamente o tramite server HTTP)
2. Il file viene identificato e trasformato in classi sorgenti Java (tramite pageCompiler)
3. Il compilatore Java compila la classe prodotta
4. La richiesta ricevuta e i relativi parametri vengono inviati all'oggetto generato (servlet), che la elabora
5. Il risultato viene inviato al client

Nel caso in cui venissero ricevute più richieste per la stessa risorsa, vengono omessi i passi 2 e 3

Le pagine JSP sono composte da tag standard HTML e tag JSP:

- Commenti

  ```
  <%-- comments.. --%>
  ```

- Direttive

  ```
  <%@ page import="java.util.*" %>
  ```

- Scriptlet

```
<% if( session.isNew() ){ <p> Welcome </p> } %>
<%! some declaration %>
<%= expression %>
```

Permettono al programmatore di accedere e memorizzare informazioni relative all'elaborazione corrente (request = HttpServletRequest, response = HttpServletResponse, out = JspWriter)

Per redirigere la pagina JSP ad una determinata URL, per separare l'elaborazione della richiesta dalla visualizzazione della risposta: <jsp:forward page="URL" />
Inserimento di una pagina identificata da una URL durante l'esecuzione: <jsp:include page="URL" flush="true" />

La pagina JSP viene trasformata in classe java

```
public void _jspService(
        HttpServletRequest req,
        HttpServletResponse res) throws IOException, ServletException;
```

Il codice di _jspService è composto da: inizializzazione e gestione degli errori e codice contenuto nello scriptlet e dalle richieste di stampa di testo sull'oggetto "out" incapsulate tra due scriptlet.

LISTENER

L'insieme dei componenti ospitati all'interno di un contenitore JavaEE è completamente gestito da questo: ne governa il ciclo di vita, ne definisce le politiche di attivazione, controlla l'uso delle risorse e della concorrenza. Il contenitore offre un insieme di meccanismi volti a permettere ad un'applicazione di reagire opportunamente ai cambiamenti degli stati dei singoli componenti. I listener permettono inizializzazione e distruzione del WebContext, modificare attributi del WebContext, creazione, modifica e distruzione di sessioni, ricezione e modifica dei parametri di una richiesta da parte del contenitore.

Si dichiara un ascoltatore degli eventi legati ad un contesto web attraverso l'annotazione @WebListener: nel momento in cui viene deployata l'applicazione, tutte le sue classi sono scandagliate alla ricerca di eventuali ascoltatori (creando un'istanza di tali classi e associando la tipo di evento che essi dichiarano di ascoltare) e quando si verifica un evento del tipo richiesto si invoca il metodo opportuno (nel contesto del thread in cui si è verificato l'evento.

IL PARADIGMA MVC

Le applicazioni web, lato server, sono costruite mediante un approccio stratificato. Ciascuno stato interagisce unicamente con gli strati adiacenti e limita le dipendenze reciproche definendo i punti di contatto sotto forma di interfacce. Questo approccio enfatizza la modularità della soluzione ed il principio della separazione delle responsabilità.

Il Data Access Layer si occupa del trasferimento delle singole entità da e verso la sorgente dei dati: offre metodi per il caricamento, il salvataggio, la cancellazione e la modifica di singoli oggetti referenziati attraverso un identificatore univoco (ID) e può anche offrire metodi per cercare oggetti sulla base del loro contenuto e per riperire tutte le entità connesse tramite una certa relazione ad un'altra entità data. Spesso è costituito da un insieme di oggetti DAO (Data Access Object) che possono appoggiarsi ad un ORM (Object Relational Mapping) o implementare le proprie logiche direttamente.

Il Domain Layer rappresenta il modello logico dei dati cui deve essere applicata la logica di business dell'applicazione: questo costituisce un'astrazione dei concetti e delle relazioni tra essi esistenti all'interno del dominio applicativo, e si appoggia allo strato di accesso dati per reperire/rendere persistenti le informazioni che lo alimentano.

Il Service Layer definisce il contratto tra l'utente e l'applicazione: offre metodi che corrispondono alle azioni logiche che l'utente può svolgere con l'applicazione. Inoltre, coordina le attività e delega i vari compiti agli oggetti del dominio applicativo: il suo stato riflette la progressione dei compiti da svolgere, piuttosto che lo stato di ciascun compito.

Il Presentation Layer ha sostanzialmente due compiti principali: trasformare le richieste provenienti dall'utente in operazioni sullo strato di servizio e incapsulare i risultati all'interno delle risposte che dovranno essere visualizzate sul dispositivo dell'utente. Data la natura multi-user delle applicazioni web, questo strato può interagire con il meccanismo delle sessioni.

La sequenza degli eventi non è nota a priori: i comandi arrivano in successione nel tempo e il programma reagisce agli eventi esterni adottando un opportuno comportamento in base alla storia passata. Quindi, occorre "dare un senso" a questa sequenza interpretando ogni evento nell'ottica della funzione che il programma intende svolgere: bisogna, ad esempio, distinguere i dati inseriti dai comandi, verificare correttezza e congruenza dei dati, eseguire comandi ecc..
Per evitare che i programmi diventino rapidamente illeggibili, occorre separarli in un insieme di componenti ed uno degli approcci più adottati è quello detto Model-View-Controller.
La vista è una rappresentazione visuale del modello. Il controllore intercetta le richieste entranti, estrae i parametri, invoca lo strato di servizio e sceglie la vista da presentare, che verrà popolata con i dati ottenuti dal modello.

MODELLO

Un'applicazione è uno strumento informatico che consente ad un utente di elaborare informazioni: esse devono essere contenute e sono soggette ad un insieme di vincoli logici. Per gestire tali informazioni si utilizza un apposito oggetto che viene detto "modello". Esso non si occupa affatto dell'interfaccia ma solo dei dati che vengono incapsulati nei suoi attributi su cui l'applicazione opera. Può essere acceduto da uno strato di servizio. E' realizzato tramite un inseme di oggetti Java memorizzati all'interno della sessione o dell'applicazione, o resi persistenti in file o in una base dati.
Il modello deve essere il punto di partenza nello sviluppo di un progetto.

VISTA

Una vista è un oggetto specializzato nel mostrare quanto contenuto nel modello. La vista non modifica in alcun modo il modello e, a parità di modello, possono esistere viste diverse, ciascuna dedicata a mostrarne una parte o a presentare le informazioni in modo diverso. Le viste sono realizzate tramite "scheletri" HTML/XML/JSON

CONTROLLORE

Oggetto che mette in comunicazione le azioni eseguite sull'interfaccia offerta dalla vista con le operazioni offerte dal modello. Il controllore costituisce il nucleo dell'interazione ed è rappresentato da codice che identifica i parametri ricevuti, aggiorna il modello attraverso le interfacce di servizio e sceglie la vista opportuna

Per ogni tipo di richiesta viene definita una URL opportuna: qui viene mappato il controllore responsabile della sua gestione (spesso implementato come servlet) e per ogni vista viene preparata una opportuna implementazione. Per ogni URL supportata, occorre definire un controllore con i relativi metodi.

SPRING FRAMEWORK

Il cuore del framework Spring è basato su Inversion of Control, Dependency Injection e programmazione orientata agli oggetti: si fa largo utilizzo del concetto di POJO-oriented development, ovvero non obbliga ad organizzare le classi in una specifica gerarchia di ereditarietà, nè ad implementare specifiche interfacce. Tutte le applicazioni ad oggetti sono basate su un insieme di classi che interagiscono secondo una qualche logica, ciascun oggetto è responsabile di ottenere i riferimenti degli oggetti con cui deve interagire, detti dipendenze. Questo pattern porta alla realizzazione di applicazioni strettamente accoppiate: se una classe dipende da una seconda non può essere compilata, testata e deployata in assenza della seconda, è ciò rende il sitema fragile e difficile da mantenere! Occorre rendere indipendenti le due classi, eliminando i riferimenti in compilazione ma facendo in modo di non perdere il legame in fase di esecuzione.

INVERSION OF CONTROL

Si elimina la dipendenza diretta nel codice sorgente introducendo un'interfaccia che ne astragga il comportamento. Essa separa due tipi di responsabilità: COSA deve essere fatto e QUANDO deve essere fatto; in generale, si realizza questo tipo di inversione passando da una modalità a chiamata diretta su un oggetto a propria scelta ad una chiamata indiretta su un oggetto fornito da altri. Questo sposta il problema su chi deve instanziare la prima classe, ossia occorre fornire un parametro opportuno al costruttore.

DEPENDENCY INJECTION

Pattern di programmazione che realizza l'accoppiamento tra gli oggetti in fase di esecuzione invece che in fase di compilazione: si affida ad un componente esterno la responsabilità di creare un grafo di dipendenze da oggetti: essi vengono tra loro collegati sulla base di un insieme di elementi di natura dichiarativa (nome, tipo, annotazioni), da una terza parte (framework) appositamente delegata.
In questo modo, tutte le dipendenze vengono create e salvate in un contenitore e successivamente iniettate nel programma principale sotto forma di proprietà degli oggetti: approccio contrario alla normale creazione di applicazioni!

Ci sono diversi modi con cui i vari componenti possono essere legati tra loro in Spring, eventualmente mischiandoli tra loro:

- Configurazione esplicita tramite un file XML
- Configurazione esplicita tramite l'utilizzo di classi Java
- Scoperta implicita dei bean e autowiring automatico

ASPECT ORIENTED PROGRAMMING

Paradigma di programmazione in grado di descrivere comportamenti trasversali all'applicazione, separandoli dal dominio applicativo. Si definisce cross-cutting concern una qualunque funzionalità che riguarda più punti di un'applicazione: esse possono essere modularizzate in speciali classi, chiamate aspect. Portano due benefici: la logica relativa ai comportamenti trasversali non è ripetuta e le classi a cui applicare gli aspeetti sono più pulite poichè contengono solo le loro funzionalità primarie.

Il lavoro compiuto da un aspect è detto advice, e Spring supporta 5 advice: before, after, after-returning, after-throwing, around (before+after). Da un punto di vista pratico, nella programmazione ad aspetti le classi base scritte dal programmatore sono modificate in fase di caricamento.

CONTENITORE

In un'applicazione Spring, gli oggetti sono creati e legati tra loro da un container che crea oggetti, li collega e ne gestisce il ciclo di vita. Vi possono essere diverse implementazioni, categorizzate in due tipologie: BeanFactory (scenari semplici) e ApplicationContext (scenari di complessità arbitraria).

Il primo (XMLBeanFactory) è un semplice contenitore fornito da Spring, che crea ed istanzia i bean insieme a tutte le configurazioni di dipendenza. Quando un bean viene richiesto, il client ne ottiene un'istanza completamente funzionale. Utilizzato solitamente per applicazioni mobili o nei contesti in cui le risorse sono limitate.

Il secondo (ApplicationContext) ne è un'estensione che, alla gestione del ciclo di vita dei bean, aggiunge diversi supporti, tra cui una versione estesa del pattern "observer" con il supporto della tecnica publih/subscribe.

BEAN

Spring identifica come bean tutte le classi che sono etichettate come @Component. Se una classe è etichettata come @Configuration, tutti i suoi metodi etichettati con @Bean vengono invocati. In una tradizionale applicazione Java, il ciclo di vita degli oggetti è semplice: viene usata la keyword "new" per istanziare l'oggetto e lo rende pronto per essere utilizzato, per poi divenire candidato per la garbage collection.

Il ciclo di vita dei Bean è più complicato:

- Il contenitore carica le definizioni dei bean. Quando gli viene chiesto un bean specifico lo istanzia
- Il bean viene popolato con le proprietà dichiarate nelle definizioni. Se una proprietà ha un riferimento ad un altro bean, questo viene creato e inizializzato prima di procedere con la creazione del bean che lo referenzia
- Se il bean implementa determinate interfacce, vengono chiamati i metodi appropriati.
- Se viene indicato un BeanPostProcessor, il framework ne invoca il metodo di pre-inizializzazione
- Se l'oggetto creato implementa l'interfaccia InitializingBean, il framework invoca il metodo afterPropertiesSet()
- Se specificato tramite l'annotazione @Bean(initMethod="..."), viene invocato il metodo indicato
- Se viene indicato un BeanPostProcessor, il framework ne invoca il metodo di post-inizializzazione
- Al termine del suo ciclo di vita, se implementa DisposableBean viene invocato il metodo destroy(). Se è stato specificato un distruttore, questo viene eseguito

All'atto della definizione, è possibile indicarne la visibilità con @Scope(...)

AUTOWIRING

L'autowiring è un modo per lasciare a Spring il compito di risolvere automaticamente le dipendenze tra i bean.

Autowire per costruttore: la presenza dell'annotazione @Autowired su un costruttore con parametri abilita l'istanziazione del bean (per ciascun parametro il contenitore cerca un bean del tipo richiesto e lo inietta nel costruttore). Se sono presenti più bean il cui tipo corrisponde a un dato argomento del costruttore, viene generata un'eccezione: per evitare ambiguità si può segnalare che un dato bean non è un candidato primario per l'iniezione utilizzando primary=false.

Autowire per tipo: l'annotazione @Autowired può essere applicata anche a singoli campi di un oggetto o ai suoi metodi setter. In questo caso, il contenitore cerca, per ogni campo/metodo, un bean il cui tipo coincide con il tipo richiesto: se è un array o una collezione, esso inietta tutti i bean con il tipo degli elementi degli array o della collezione.

Autowire per nome: usando l'annotazione @Resource(name="...") è possibile indicare che deve essere iniettato uno specifico bean sulla base del nome con cui è noto al contenitore.

APPLICATION.PROPERTIES

Ogni progetto Spring Boot ha associato un file di configurazione application.properties il cui caricamento è implicito. Si possono aggiungere righe nella forma <chiave>=<valore> che dipendono dai moduli aggiunti al progetto. I valori fissati all'interno del file delle proprietà possono essere assegnati a campi presenti all'interno della classe principale o in qualunque altra classe etichettata con l'annotazione @Component

VISUALIZZAZIONE PAGINE HTML

Una classe annotata con @Controller viene automaticamente istanziata in quanto @Component (Spring cerca tra i suoi metodi @RequestMapping per identificare quali URL siano disponibili). Ogni volta che viene ricevuta una richiesta per il verbo HTTP e URL associati ad un dato metodo, questo viene invocato ed il suo valore di ritorno viene utilizzato per selezionare una vista opportuna che verrà restituita come risposta. La trasformazione di tale valore in una vista è affidata ad un apposito bean, il viewResolver e se tale valore è una stringa, il suo nome indica il nome della vista.
Il file selezionato viene elaborato ed inviato come risposta alla richiesta e il controllore può specificare, tra i propri parametri, un riferimento ad un oggetto di tipo Model (consiste sotanzialmente in una mappa chiave-valore). Un controllore può ricevere informazioni in tre modi: tramite i segmenti di testo che formano la URL (annotare con @PathVariable), tramite i parametri posti nella queryString (parametri del metodo del controllore, usando annotazione @RequestParam<paramName>) oppure tramite POST e PUT (passati sotto forma di oggetto Java a condizione che sia dotato di metodi get/set, e può venire indicato come parametro in ingresso al metodo del controllore e ci pensa Spring a istanziare).

In molte situazioni è necessario verificare che i dati rispettino un insieme di vincoli formali, e Spring offre un meccanismo generale: gli attributi dell'oggetto DTO possono essere etichettati con annotazioni tipo @NotNull @Min<val> @Valid. Quest'ultimo aggiunge un parametro BindingResult che verifica gli errori che possono essere visti tramite hasErrors().

SPRING SERVICES

Le funzionalità offerte dall'applicazione sono normalmente formulate sotto forma di API: i metodi di tale interfaccia costituiscono la logica mentre i data passati/restituiti descrivono le astrazioni del dominio applicativo. A partire dalla definizione di servizio, è possibile formulare un insieme di test e un'implementazione fittizia (mock) che permette di suddividere il lavoro tra gruppi di sviluppatori diversi.
Il meccanismo di inserimento delle dipendenze offre il substrato per implementare facilmente la logica applicativa: appositi oggetti etichettati con @Service offrono le funzionalità effettive del servizio che si intende erogare.

Il ViewModel è un oggetto che veicola i dati da mandare alla vista (JSON), tipicamente contiene i valori inseriti dall'utente nei form.
I DTO (Data Transfer Object) modella le informazioni che l'interfaccia del servizio offre e/o consuma: il controller si occupa di trasformare le informazioni ricevute in ingresso in uno o più DTO necessari per il servizio e si occupa inoltre di trasformare i dati restituiti da questo in un ViewModel adatto.
Le Entity modellano i dati così come vengono tenuti nel DB, permettendo di disaccoppiare lo strato di servizio da quello di persistenza e aggiungere informazioni necessarie alla indicizzazione e alle query.

Per accedere ai dati si può non coinvolgere il framework Spring (totale flessibilità, ma nessun servizio offerto dalla piattaforma!) oppure appoggiarsi alle funzionalità del framework (sforzo leggermente maggiore nel configurare l'ambiente ma ben ripagato!)

REPOSITORY

Nome usato nella terminologia Spring per definire una classe DAO, che offre metodi per eseguire operazioni di accesso ad una singola tabella della base dati sottostante, e si definisce come Repository<T, Id>.
T rappresenta la classe che descrive il dato mentre Id è il tipo di chiave primaria degli oggetti T (deve essere serializzabile e in T deve esserci un campo annotato con @Id).

JDBC

Java Data Base Connectivity: permette l'accesso a database relazionali, con un basso livello di astrazione (il programmatore si deve occupare di quasi tutti i dettagli).
I driver JDBC rappresentano il punto di congiunzione tra le due librerie e il DBMS: esse vengono istanziate indirettamente attraverso la classe factory DriverManager. Il DB è rappresentato da una URL e per connettersi bisogna usare getConnection(DbURL, User, Psw). Il formato è solitamente del tipo jdbc:subprotocol:source.

Per eseguire un'interrogazione, bisogna effettuare cinque passi fondamentali:

1. Registrazione del driver (una sola volta)
2. Connessione al database
3. Invio dell'interrogazione
4. Lettura dei risultati
5. Rilascio delle risorse

L'interfaccia per inviare i comandi sotto forma di stringhe è chiamata Statement, che ha come metodi quelli di eseguire delle interrogazioni (executeQuery, executeUpdate) e rilasciare le risorse utilizzate (close()).
Come oggetto risultato si ha il ResultSet, che organizza i dati in forma tabulare (ad ogni riga corrisponde un record restituito, in modo tale da poter richiamare le righe ad una ad una, tramite un cursore). Si accede ai dati della riga

puntata dal cursore tramite i metodi getXXXX(fieldName) o getXXXX(columnIndex).

E' possibile associare una classe ad ogni tabella presente nella base dati: i suoi attributi quind corrispondono alle colonne della tabella. Per ogni classe del dominio si definisce anche una classe DAO (Data Access Object) i cui metodi offrono le funzionalità necessarie a creare, reperire, modificare ed eliminare singole righe nella tabella (CRUD). Per default, eventuali cambiamenti vengono automaticamente confermati dopo l'esecuzione di ogni istruzione SQL e per disabilitare questo comportamento (per raggruppare più istruzioni in una transazione oppure gestire la concorrenza) si utilizza il metodo setAutoCommit(false).

HIBERNATE E JPA

L'approccio ad oggetti si presta molto bene a rappresentare sistemi software complessi ma manca un meccanismo generalizzato per rendere persistenti le informazioni. Per superare il disadattamento di impedenza si può utilizzare un ORM (Object-Relational Mapping), ossia uno strato software che converte i tipi di dato da un formato ad oggetti ad uno relazionale e viceversa. Ad ogni riga del DB si associa un oggetto di classe opportuna (introducendo un meccanismo per cui oggetto e riga si mantengono sincronizzati, almeno in momenti specifici).

Un ORM è un approccio concettuale alla modellazione, interrogazione e trasformazione dei dati che rende trasparenti le operazioni si conversione e rappresentazione a basso livello delle informazioni all'interno della base dati. Internamente genera le istruzioni necessarie per accedere alla base dati garantendo:

- Produttività: viene eliminata la scrittura di codice altamente ripetitivo
- Manutenibilità: meno codice è più facile da mantenere
- Prestazioni: il caricamento delle entità collegate da una relazione avviene solo se richiesto
- Astrazione: abbiamo indipendenza dal produttore del DBMS

Che limiti abbiamo? In caso di cancellazioni massive o esecuzione di join con un elevato numero di elementi ci possono essere problemi di prestazioni e il suo utilizzo massivo può condurre a basi dati mal progettate.

Un ORM per Java è Hibernate, adatto sia per applicazioni stand-alone che per applicazioni web. Ad ogni entità presente nel modello logico è associata una classe Java (dotata sia di getter/setter che di un ID). Le entità sono associate a tabelle relazionali mediante un file XML oppure mediante annotazioni. Nel primo caso la sintassi non è semplice facile da leggere o da scrivere, e inoltre bisogna mantenere la sincronizzazione tra file XML e classe Java. Nel secondo caso ogni classe che definisce un'entità viene preceduta dall'annotazione @Entity e l'attributo chiave è preceduto da @Id; se il nome di un attributo non coincide con il nome della corrispettiva colonna si può usare @Column(name="<nome>").

Hibernate permette che una chiave primaria semplice abbia un tipo primitivo: se si intendono usare chiavi assegnate dal DBMS, è bene non usarli, ma indicare che è obbligatoria e non può essere modificata in seguito.
Se una chiave è annotata con @GeneratedValue, è appunto compito del DBMS assegnarle un valore:

- AUTO usa la strategia migliore in base al dialetto SQL
- IDENTITY usa un valore numerico incrementato automaticamente legato alla specifica colonna ma può impattare sulle prestazioni
- SEQUENCE genera un valore ma non è specifico della colonna, per cui viene incrementato anche se non si sta inserendo nulla
- TABLE seleziona il nuovo valore da una apposita tabella nel DBMS

Se il tipo della chiave non è numerico ma UUID, hibernate genera il valore a livello applicativo.
Per ogni proprietà presente in un oggetto è possibile definire la colonna della base dati corrispondente, il tipo di SQL da adottare e i vincoli su esistenza e unicità; il tipo degli attributi viene utilizzato per comprenderne la mappatura sulla base dati mentre i riferimenti (diretti o indiretti che siano) ad altre entità sono considerati relazioni.

Se un'entità è legata ad un'altra tramite un'associazione di cardinalità superiore ad uno, essa offrirà, nella classe

Java, una proprietà di tipo Collection, e nel file di mapping verrà riportata la corrispondente associazione esprimendone la cardinalità e le eventuali tabelle di appoggio. Si può navigare la relazione in entrambe le direzioni, per cui essa dev'essere definita in entrmbe le entità: il discorso vale anche per l'aggiornamento/cancellazione, e si risolve utilizzando @cascade (PERSIST, UPDATE, REMOVE, ALL).
Un'entità deve avere un costruttore pubblico, non deve essere final ed i campi che devono essere resi persistenti devono essere privati, protetti o accessibili solo al package (e devono essere serializzabili per poterli gestire in modalità detached).

Il file hibernate.cfg.xml fornisce le informazioni necessarie per accedere ad una specifica base dati. La proprietà hbm2ddl.auto indica quali azioni eseguire in inizializzazione. La proprietà show_sql abilita la stampa sul log di default delle istruzioni SQL che vengono eseguite da hibernate.
Dal punto di vista della persistenza, un oggetto Java istanza di una classe mappata, può assumere tre stati differenti:

- Transient: appena creato da una new, e quindi senza nessun legame di persistenza
- Persistent: eventuali modifiche vengono rese persistenti alla chiusura della transazione
- Detached: dopo la chiusura della transazione, l'oggetto perde il legame con il contesto di persistenza

Hibernate offre due metodi di accesso all'istanza: load (viene creato un proxy che prova ad accedere, altrimenti EXCEPTION) oppure get (si interroga subito, altrimenti NULL).

JPA (Java Persistence Annotation) è un'interfaccia standard per rendere persistenti oggetti tramite ORM e viene definita a partire dal modello hibernate (si configura da META-INF/persistence.xml al posto di hibernate.cfg.xml), ma in Spring Boot il framework provvede a simularne l'esistenza.

REST

L'evoluzione di internet impone la realizzazione di sistemi distribuiti scalabili:

- Eterogeneità: capacità di fare interoperare dispositivi diversi in termini di risorse hardware, SO, linguaggi di programmazione ecc..
- Scalabilità: capacità di garantire il mantenimento di livelli di servizio al crescere del numero di utenti/-transazioni
- Evoluzione: proprietà di un'architettura di non introdurre vincoli strutturali che impediscono la manutenzione evolutiva di un servizio
- Resilienza: capactià di un sistema di mantenere i propri livelli di servizio anche a fronte di malfunzionamenti locali
- Efficienza: misura del rapporto tra le risorse impegnate per offrire un servizio e il valore del risultato prodotto
- Prestazioni: valutazione assoluta del tempo/risorse necessarie per erogare un servizio

REST (Representational State Transfer) è uno stile con il quale realizzare architetture distribuite. Client-Server privo di stati (implementazioni semplici e scalabili). Più servizi ospitati su un server: ognuno gestisce un insieme di "informazioni" che hanno un nome univoco su scala globale (URI) e supporta operazioni elementari di tipo CRUD. Le informazioni associate ad una URL possono essere relative a singole voci, collezioni di voci o risultati di computazioni funzionali.

Poichè ogni entità ha un nome univoco, è possibile legarle insieme specificando il nome della relazione. Il protocollo HTTP costituisce un ambiente naturale nel quale le richieste di tipo CRUD possono essere mappate sulle caratteristiche del trasporto (ogni richiesta consiste di un'azione e di una URL).
Ogni elemento può essere rappresentato in formati alternativi (JSON, XML, CSV...) e HTTP dispone di un meccanismo nativo per negoziare il formato di dati scambiati. Il grado di adesione di un dato progetto ai principi generali dell'architettura REST può essere catalogato sulla base del modello Richardson:

- LIVELLO 0: HTTP viene usato come protocollo di trasporto, tutte le richieste sono indirizzate ad una singola URL e si usa prevalentemente il metodo POST.

- LIVELLO 1: ogni entità gestita dal sistema dispone di una propria URL e si usa prevalentemente il metodo POST
- LIVELLO 2: i comandi da eseguire vengono mappati sui metodi HTTP e gli esiti dell'operazione sono espressi tramite i codici di stato
- LIVELLO 3: Per ogni oggetto trasferito, il server include una serie di collegamenti (URI) che indicano quali azioni possano essere compiute sull'oggetto stesso

L'uso di URL differenti (livello 1) affronta il problema della gestione della complessità attraverso il paradigma "divide et impera". L'uso dei metodi HTTP e dei codici di stato associati (livello 2) offre un meccanismo di standardizzazione, eliminando variazioni superficiali che sono fonte di inutile complessità. L'uso di collegamenti ipertestuali (livello 3) permette la scoperta dinamica dei servizi, rendendo le interfacce più auto-documentanti e aprendo la strada alla realizzazione di microservizi.

Quando l'architettura di un'applicazione distribuita è basata su un insieme di componenti lato server indipendenti tra loro, si parla di microservizi. Ogni componente viene eseguito nel proprio processo/container/macchina e coopera con gli altri utilizzando protocolli di rete. Essi possono quindi essere deployati indipendentemente l'uno dall'altro facilitando la scalabilità e la manutenibilità del sistema. I benefici di utilizzare questo tipo di servizi sono evidenti, ma bisogna tenere conto che in un ambiente distribuito le interazioni sono asincrone e possono fallire, dando origine a pattern di programmazione più elaborati (con debugging altrettanto complesso); inoltre, il lavoro dei sistemisti cresce, dovendo mantenere un gruppo di processi che possono richiedere manutenzione e riavii su base regolare.

Tra i fondamenti del modello REST c'è l'assenza di stato: ogni richiesta da un client ad un server, quindi, deve contenere tutte le informazioni necessarie per comprendere la richiesta e non deve basarsi su informazioni di contesto memorizzate sul server. Ogni oggetto trasferito porta con sè il proprio stato, e non sono perciò legati ad una sessione, avendo nell'oggetto tutto quanto necessario per operare su di esso: ciò abilità la scalabilità orizzontale. Come si gestisce allora l'identità del client?

- Basic Authentication: header di tipo Authorization il cui valore inizia con basic ed è seguito da una codifica reversibile della password
- HMAC: header di tipo Authorization il cui valore inizia con hmac e contiene un riassunto del metodo HTTP, della URL, della data/ora corrente e, se presente, del corpo della richiesta
- OAUTH2: header di tipo Authorization il cui valore inizia con bearer seguita da un token autorizzativo univoco ottenuto dal server

In Spring, si ha l'annotazione @RestController: tutti i metodi definiti in una classe con essa possono mettere a disposizione un endpoint del servizio: per effettuare il passaggio dei parametri nel corpo della richiesta si utilizza @RequestBody. Con @PathVariable è possibile inserire porzioni di URL come parametri del metodo.

Spring offre due metodi per negoziare il contenuto: Content Negotiation in cui è compito del programmatore fornire diversi ViewResolver in funzione dei tipi di richiesta che si vuole servire oppure Message Conversion, per cui non c'è necessità che il Dispatcher Servlet gestisca il passaggio da model a view e viceversa (il Dispatcher Servlet considera l'Accept Header della richiesta e cerca il MessageConverter adeguato).
Con il primo metodo, non bisogna modificare il codice del controller mentre il secondo è il modo più diretto e conveniente quando si sviluppa esclusivamente REST.

SPRING SECURITY

Sebbene sia possibile inserire, a livello applicativo, vincoli di sicurezza, non è una buona idea farlo, poichè essa deve essere implementata in modo dichiarativo, lasciando all'ambiente di esecuzione il compito di tradurre le specifiche in implementazioni concrete in tutti i punti necessari. Spring Security fa esattamente questo: a livello web tramite filtri e a livello d'invocazione dei metodi tramite "aspetti" che specificano chi abbia diritto di accedere. Questo implica la gestione di una base dati degli utenti, che supporti i processi di registrazione, autenticazione ecc.. e la presenza di un meccanismo di autorizzazione che consenta l'accesso una volta autenticati.

Gli endpoint di registrazione e accesso non sono soggetti a restrizioni, ma bisogna effettuare numerose verifiche di correttezza dei dati inviati. Il modulo di Spring Security si occupa di creare un'infrastruttura configurabile per la gestione dichiarativa degli accessi e dei ruoli: è possibile esprimere restrizioni in due modi, a livello URL e a livello di singoli metodi di Bean.

La classe HttpSecurity offre una fluent API per definire in modo compatto le URL da proteggere, a parte /logout che termina la sessione corrente tramite una POST. Per default, tali sessioni si appoggiano a quelle di HTTP per conservare le informazioni di accesso e se la richiesta non risulta appartenere ad una sessione viene lanciata un'eccezione AuthenticationException. E' possibile etichettare i singoli metodi offerti da un servizio con annotazioni legate alla sicurezza, usando @PreAuthorize, @PostAuthorize, @PreFilter, @PostFilter.
Nel caso di servizi REST, non ha senso reindirizzare le richieste non autenticate. Utilizzando il meccanismo di autorizzazione JWT (JSON Web Token) abbiamo che il client invii, nella propria richiesta, una intestazione formate da tre parti separate da '.':

- <header>: metadati sul tipo di token
- <payload>: informazioni relative all'utente ed ai permessi riconosciuti
- <signature>: contiene una firma delle parti precedenti

Tali parti sono codificate in Base64.

ARCHITETTURE

Un'architettura è l'insieme delle strutture di alto livello che formano un sistema software e delle relazioni implicite ed esplicite che intercorrono tra loro. Un'architettura software è conseguenza di scelte guidate dai casi d'uso e dai requisiti di business legati alla valorizzazione dell'artefatto prodotto: essendo il risultato dell'interazione di molteplici persone, quindi occorre che il sistema realizzato sia suddivisibile in componenti che godano di due proprietà fondamentali: basso accoppiamento e alta coesione.

L'accoppiamento è il grado di dipendenza di un componente dai dettagli implementativi di un altro componente: minore è il valore, maggiore sarà la possibilità di sostituire un dato componente con un altro dotato della stessa interfaccia ma di qualità diversa (il processo di refactoring è normalmente volto a ridurre l'accoppiamento di un componente ripensandone la struttura interna).

La coesione è il grado di focalizzazione di un componente software su un dato compito: un componente ad alta coesione contiene funzionalità altamente correlate tra loro e non ulteriormente separabili.
Qualità logiche di un'architettura:

- Estensibilità: misura la facilità di modificare specifiche e requisiti esistenti o aggiungerne di nuovi
- Testabilità: misura la facilità con cui è possibile mettere alla prova, in modo automatico, i singoli componenti e il sistema nel suo complesso
- Comprensibilità: misura la facilità con cui uno sviluppatore "standard", che si approccia per la prima volta al sistema riesce a comprendere dinamiche e meccanismi presenti al suo interno
- Resilienza: capacità di un sistema di restare in funzione nonostante la presenza di malfunzionamenti parziali
- Sicurezza: grado di difficoltà necessario a violare i meccanismi di protezione di un sistema per trarne beneficio

Qualità operative di un'architettura:

- Scalabilità: la capacità di un sistema di fare fronte a carichi crescenti di lavoro aumentando la quantità di risorse ad esso allocate
- Manutenibilità: il grado di facilità con cui gli operatori di un sistema possono osservarne il comportamento, diagnosticarne malfunzionamenti ed intervenire al suo interno per riportarlo in servizio
- Installabilità: la facilità con cui è possibile mettere in campo un prodotto

- Usabilità: il grado di efficacia, efficienza e soddisfazione provato da un utente nell'usare il sistema
- Disponibilità: la percentuale di tempo in cui un dato sistema è effettivamente operativo e funzionante rispetto all'intervallo di tempo in cui la sua funzionalità è attesa

I 12 FATTORI

La disponibilità di piattaforme cloud in grado di ospitare applicazioni di terze parti in modo scalabile e facile da deployare e da mantenere ha portato la comunità degli sviluppatori ad elaborare una serie di principi architetturali e metodologici relativi alle applicazioni web:

1. Codebase: un progetto deve avere un'unica base di codice, gestita da un sistema in grado di tracciare le revisioni e di esporre un manifesto che ne elenchi in modo esplicito le dipendenze, da cui sia possibile effettuare messe in campo multiple; per supportare la collaborazione tra sviluppatori e gestire le versioni. Git e Maven offrono gli strumenti di base per supportare tale principio
2. Dipendenze: le dipendenze devono essere dichiarate in modo esplicito ed isolate dal resto del codice; per rinforzare la consistenza tra versioni di sviluppo e di produzione, semplificare l'impostazione di nuove applicazioni e supportare la portabilità tra piattaforme cloud. Si può usare MavenCentral o NodeJs
3. Configurazione: i dettagli di configurazione devono essere presenti nell'ambiente di esecuzione e non all'interno dei codici sorgenti; per modificare il comportamento senza toccare il codice, semplificare il deploy e ridurre il rischio di rivelare credenziali.
4. Servizi di supporto: devono essere trattati come risorse dell'applicazione; per fornire un accoppiamento lasco tra servizi e applicazione e poter aggiornare/sostituire i servizi senza toccare il codice. Si deve fare uso di componenti standard e/o custom dall'interfaccia ben nota.
5. Costruzione, Rilascio, Esecuzione: le tre fasi del ciclo di vita del software devono essere ben separate. Costruzione è dove il codice sorgente viene compilato e impacchettato in un singolo artefatto, detto build. Rilascio è dove il file così ottenuto viene combinato con le informazioni di configurazione e diventa adatto ad essere eseguito (ogni rilascio ha un identificatore univoco). Esecuzione è quando l'applicazione rilasciata viene eseguita in uno specifico ambiente. Per impedire cambiamenti in fase di esecuzione e ridurre il numero di fallimenti inattesi. Offerto da Spring Boot
6. Processi: l'applicazione viene eseguita come uno o più processi privi di stato: l'unica forma di persistenza deve essere fornita da un apposito servizio di supporto (connessa in fase di esecuzione) e nessun dato viene salvato sul file system; per ridurre la complessità di deploy, essere eseguiti in un container e migliorare la scalabilità. Si può fare salvando i dati legati allo stato su un servizio di supporto, evitando di usare filesystem e sessioni. Si può verificare accendendo e spegnendo l'applicazione e vedendo se si sono persi dei dati.
7. Porte: i servizi offerti da un'applicazione conforme ai principi vengono esposti su apposite porte di rete: non deve essere necessario deployare tali servizi all'interno di un servlet container o di qualcosa di analogo; per aumentare il disaccoppiamento e la portabilità dei processi. Si può fare usufruendo di container e ambienti di configurazione che automatizzano l'esposizione di singole porte all'insieme degli host coinvolti
8. Processi: si gestisce la scalabilità di una applicazione istanziando più processi della stessa (su macchine diverse). I singoli processi possono anche adottare tecniche multithreading o modelli asincroni ad eventi, ma la scalabilità su macchine diverse resta la risorsa principale; per aumentare la scalabilità orizzontale. Si può fare attraverso reverse-proxy ed altri pattern architetturali che favoriscono il bilanciamento del carico.
9. Terminabilità: i processi devono poter essere avviati velocemente e fermati senza problemi (gestendo il rilascio delle risorse da cui dipendono): per facilitare il test e la scalabilità, consentire la messa in campo rapida di una nuova versione e fornire robustezza alla produzione.
10. Parità sviluppo-produzione: gli ambienti di sviluppo, test e produzione dovrebbero essere il più possibile simili tra loro; per ridurre il rischio di periodi di arresto forzato e supportare l'integrazione continua. Si può fare appoggiandosi ad un prodotto.
11. Log: i log vanno trattati come flussi di eventi e non gestiti direttamente dal codice sorgente (compito dell'ambiente di esecuzione); per spostare sulla piattaforma l'onere del logging e migliorare la flessibilità nell'osservare il comportamento nel tempo. Viene offerta l'infrastruttura di log dai framework

12. Processi di amministrazione: i compiti amministrativi vanno eseguiti come processi una tantum; per ridurre gli errori nella manutenzione/evoluzione delle applicazioni. Esempi: popolamento iniziale di una base dati, migrazione di una repository.

Aderendo ai 12 fattori, un'applicazione diventa in grado di essere facilmente automatizzata, essere portata in ambienti di esecuzione differenti e messa in campo su un sistema cloud, oltre ad essere soggetta ad un processo di integrazione e messa in campo continui e ad essere scalabile senza particolari cambiamenti.

REAL TIME APPLICATIONS

Le piattaforme web offrono un meccanismo semplice per offrire servizi a comunità di utenti grazie al paradigma client/server, ma vi sono casi in cui ocorre ricevere aggiornamenti in tempo reale da parte del server. Storicamente sono stati messi a punto meccanismi differenti con vari limiti e tutti condividono il principio per il quale la comunicazione deve essere iniziata dal lato client.

- Polling semplice: i client fanno richiesta di informazioni al server e questo risponde immediatamente. Non scala all'aumentare del numero dei clienti, inoltre iniziare una nuova connessione ha un elevato costo (7 pacchetti IP), oltre al fatto che molte richieste sarebbero come un attacco DDOS
- Long Polling: meccanismo di richiesta con risposta potenzialmente ritardata. Se il server non è in grado di fornire le informazioni richieste, ferma la richiesta e la riutilizza per quando le info sono disponibili. Alla ricezione di una risposta, il client esegue subito una nuova richiesta, mentre se ha bisogno di mandare dati mentre aspetta, allora apre una connessione parallela. Per poter gestire questo tipo di situazioni, lo strato applicativo sul server mantiene, per ogni possibile client, una coda di messaggi, la quale viene controllata ad ogni richiesta del client e, nel caso, inoltrando tutti i messaggi ad esso destinati, chiudendo la risposta. Nel caso in cui la coda fosse vuota, la comunicazione viene tenuta aperta per un lasso di tempo, che se scade senza richieste, fa chiudere la connessione, altrimenti viene inserito nella coda e immediatamente recapitato usando la connessione in sospeso
- Streaming: modello simile al long polling, basato sulla codifica di trasferimento chunked. I client aprono una connessione verso il server che configura la risposta come Transfer-Encoding: chuncked; tale risposta non viene mai terminata dal server che si limita ad inviare, di volta in volta, nuovi chunk effettuando il flush della comunicazione. Via via che i dati vengono trasferiti al client, questo può elaborarli ma non tutti i browser supportano correttamente tale meccanismo

I server web utilizzano solitamente un modello di concorrenza adatto a gestire molte richieste di breve elaborazione, pushando e poppando thread da un threadpool, ma l'utilizzo di richieste che durano a lungo nel tempo vincola tali thread portando rapidamente all'esaurimento delle risorse disponibili.

La specifica JavaEE6 ha introdotto la possibilità di dichiarare un servlet in modalità asincrona, per cui la richiesta viene incapsulata in un oggetto di tipo AsyncContext lasciando libero il thread nel caso in cui non ci siano le informazioni per generare la risposta. E' possibile fissare anche un limite di tempo di elaborazione oltre il quale la connessione deve essere chiusa. Per funzionare la classe del servlet deve essere annotata con @WebServlet(asynchSupported=true).

Gestire richieste asincrone presuppone che il server disponga di un meccanismo per organizzare e distribuire i messaggi, in quanto le richieste possono arrivare da fonti diverse ed essere elaborate da thread differenti, occorre adottare politiche di sincronizzazione he evitino le interferenze senza penalizzare le prestazioni (occhio ai lock globali!!).

Web Socket è un protocollo di comunicazione bidirezionale asincrono instaurato tra un client e un server con una richiesta HTTP/Upgrade ed offre una valida alternativa al long polling. Il protocollo è basato su una connessione TCP che viene tenuta aperta per tutta la durata della sessione di lavoro; presuppone lato client l'adozione di un modello a pagina singola, all'interno del quale è presente codice JavaScript che sovraintende tale connessione. L'uso di una connessione permanente ottimizza l'utilizzo delle risorse di rete e la comunicazione è full-duplex (crittografata, volendo, con TLS).
Pressochè tutti i browser supportano tale protocollo ed è lo standard per le implementazioni JavaEE7.

Una connessione web socket attraversa due fasi: la negoziazione iniziale (handshake) e lo scambio di dati. La prima inizia con una GET dal client indicando "websocket" + versione protocollo + chiave di sessione. Lo scambio di dati, invece, consente l'invio di contenuti testuali (utf-8), binari e di controllo e possono avere dimensioni arbitrarie. La classe Session modella il canale di comunicazione (istanza passata come parametro ai metodi che ne riportano gli eventi) mentre la ricezione dei messaggi è regolata dalle annotazioni poste sui metodi della classe che implementa il web socket (usando @OnMessage, massimo 3). Ogni volta che si instaura una nuova connessione, il contenitore crea una nuova istanza della classe annotata come endpoint, mentre Session offre una mappa (String-> Object, accessibile mediante getUserProperties()) in cui si possono salvare ulteriori informazioni. Sebbene il ciclo di vita di un web socket sia sostanzialmente indipendente da quello della pagina che ne ospita la componente client, è possibile creare un punto di contatto nel momento in cui la comunicazione viene instaurata (EndpointConfig accede a HandshakeRequest che accede a HttpSession).

Tramite le annotazioni, si possono anche inviare dati strutturati, specificando encoders e decoders utili alla codifica/decodifica in tali dati. Inoltre, spesso risulta conveniente utilizzare messaggi differenti per diversi tipi di eventi, e la codifica JSON si presta benissimo a questa casistica, seppur l'implementazione del decoder diventi leggermente più complessa (bisogna prima verificare ci sia un reale JSON e che, nel campo specifico, ci sia un valore lecito).
La logica con cui si collegano produttori e consumatori può essere molto complessa, per cui serve un componente in grado di orchestrare i messaggi ed i relativi protocolli applicativi. Un message broker è specializzato nel gestire la validazione e l'instradamento di messaggi originati da programmi diversi verso i rispettivi proprietari, introducendo un livello di disaccoppiamento tra mittente e destinatario e rendendo quindi più robusto il sistema. Un esempio di message broker è STOMP: divide i messaggi in categorie (connessione, transazioni, conversazione). Tutti i messaggi scambiati con un broker possono essere diretti ad una coda ed è poi lui che si occupa dell'inoltro (sia esso one-to-one o one-to-many). Infine, i messaggi possono essere storicizzati per consentire ai client di fare accesso al contesto passato.

DOCKER