ALMA MATER STUDIORUM - UNIVERSITA' DI BOLOGNA

CAMPUS DI CESENA

SCUOLA DI SCIENZE

CORSO DI LAUREA IN SCIENZE E TECNOLOGIE INFORMATICHE

UN WEBSERVICE PER COLLEGARE IL CITTADINO CON LA PROPRIA SITUAZIONE DEBITORIA

Relazione finale in

Paradigmi di Programmazione

Relatore:

Prof. Vittorio Maniezzo

Presentata da:

Giacomo Casanova

Sessione

Anno Accademico 2015/2016

SOMMARIO

1 INTRODUZIONE

Il progetto di tesi tratta di un web service, capace di reperire informazioni da un database centrale, di un programma gestionale per dare tali informazioni ad un portale che ne fa richiesta, utilizzando richieste SOAP.
Il portale ha bisogno di uno strumento che restituisca le informazioni richieste e che riceva solo dei determinati dati, opportunamente, filtrati e lavorati.
Sia il portale che il gestionale sono già esistenti e fanno parte della stessa azienda.
Lo scopo della tesi è fare vedere quei dati nel portale in maniera chiara, semplice, veloce ed ottimale, ma partiamo per ordine.

1.1 CHE COSA È UN WEB SERVICE

Un Web service è un'applicazione/un sistema software in grado di fornire uno o più servizi di un applicazione comunicando su di una medesima rete tramite il protocollo HTTP.
Un Web service consente quindi alle applicazioni che vi si collegano di usufruire delle funzioni che mette a disposizione.
Un Web service è in grado di offrire un'interfaccia software assieme alla descrizione delle sue caratteristiche, cioè è in grado di farci sapere che funzioni mette a disposizione (senza bisogno di conoscerle a priori) e ci permette inoltre di capire come vanno utilizzate. Ciò significa che con una semplice connessione ad un web service, anche senza conoscerlo (parliamo del codice), possiamo stabilire le operazioni che fornisce e possiamo subito iniziare ad usarle perchè ogni operazione ha una sua descrizione comprendente i parametri che si aspetta di ricevere, quelli che restituirà ed il tipo di entrambi.

Un web service usa HTTP, questo protocollo si occupa di mettere in comunicazione il web service con l'applicazione che intende usufruire delle sue funzioni.

Oltre ad HTTP però, il web service utilizza molti altri standard web, tutti basati su XML, tra cui:

- XML Schema
- UDDI(Universal Description,Discovery and Integration)
- WSDL (Web Service Description Language)
- SOAP (Simple Object Access Protocol)

È importante sottolineare che XML può essere utilizzato correttamente tra piattaforme differenti (Linux, Windows, Mac) e differenti linguaggi di programmazione.

XML è inoltre in grado di esprimere messaggi e funzioni anche molto complesse e garantisce che tutti i dati scambiati possano essere utilizzati ad entrambi i capi della connessione.

Quindi si tratta di uno strumento molto versatile, potente ed utile.

1.2 L'OBBIETTIVO

Il progetto nasce per l'esigenza di dare, la possibilità ad un contribuente di vedere la propria storia coattiva di un verbale/accertamento/fattura non pagata in passato e per cui non è ancora presente un pagamento.

Innanzitutto trattiamo della coattiva.

È un'attività gestionale, che parte dall'acquisizione della lista di carico (elenco debitori e/o nota di addebito) fino all'espletamento della procedura per la riscossione coattiva , nonché l'incasso ed il riversamento nelle casse comunali delle somme riscosse, con puntuale rendicontazione.

Se qualcuno non paga le multe o le fatture/accertamenti al comune allo scadere di tali atti, si passa nella fase coattiva ovvero un processo legale con varie fasi per cercare di ottenere il dovuto non ricevuto.

La fase coattiva di un atto, nell'azienda in cui è stato creato questo servizio, permette la riscossione di vari tributi locali, per esempio ICI, IMU, tassa rifiuti, multe non pagate (divieti di sosta, multe autobus/treni, bollo non pagato), violazioni del Codice della Strada;

Un comune oppure un ente comunica le sue posizioni per cui non hanno ricevuto la somma dovuta e affidano la gestione della riscossione ad un'azienda specializzata in modo da recuperare, quanto dovuto.

Questa operazione è resa possibile grazie ad un gestionale esistente, che gestisce tutte le pratiche dalla A alla Z e l'anagrafica del contribuente ad esso collegati.

La società di riscossione agisce come se fosse Equitalia, l'unica differenza è che in questo caso l'azienda che gestisce la riscossione è privata.

Fino ad oggi non vi era la possibilità di visionare la propria situazione di quanto non pagato online e da qui nasce l'esigenza, di creare un portale che interfacci i dati già presenti nel gestionale dell'azienda, visualizzando solo alcuni dati e solo al contribuente che ha ricevuto un atto della coattiva.

1.3 COME FUNZIONA IN BREVE IL GESTIONALE DELLA RISCOSSIONE

La prima fase è il caricamento di un file di testo chiamato ruolo 290 (formato standard) dentro il gestionale che poi creerà le ingiunzioni.

Il programma genererà degli elenchi con le posizioni scartate durante la procedura di carico per mancanza di dati utili, quali partita iva, codice fiscale, indirizzo ect. mentre per i rimanenti dati corretti, il gestionale produrrà le ingiunzioni.

Ogni atto, cartolina raccomandata che il gestionale crea e successivamente spedisce verrà sempre archiviato digitalmente.

I documenti, una volta stampati e pronti per la spedizione, vengono consegnati alle poste con la relativa distinta e recapitati ai destinatari.

Le ingiunzioni non notificate tramite posta per irreperibilità o perché i destinatari sono sconosciuti all'indirizzo, ritornano al mittente che dovrà verificare gli indirizzi di residenza, per

le persone fisiche mediante l'inoltro di fax presso gli uffici anagrafici nazionali, mentre per le persone giuridiche tramite ricerca nella banca dati a disposizione dell'Ente.

Ogni ingiunzione creata deve essere notificata correttamente, per potere proseguire con le fasi successive: cautelari ed esecutive.

Il contribuente può fare ricorso e ottenere un discarico o chiedere un piano di rate per il proprio atto, il gestionale gestisce anche queste procedure particolari.

Le ingiunzioni non pagate entro 30 gg. dalla data di notifica, di importo inferiore a €. 1.000 e riferite a ruoli dati in carico dopo l'entrata in vigore della legge di stabilità del 24/12/2012, diventeranno atti successivi chiamati "sollecito di pagamento".

Il sollecito di pagamento è una semplice comunicazione ricognitiva del debito inviata tramite posta ordinaria.

Il flusso, una volta generato, verrà gestito con la stessa procedura indicata per la stampa degli atti ingiuntivi.

Solo dopo 120 giorni dall'invio del sollecito, il gestionale potrà generare gli atti successivi in questo caso preavvisi di fermo o pignoramenti, in alcuni casi può essere generato anche un precetto.

La generazione dei preavvisi di fermo avviene in maniera automatica mediante lo scarico dalla banca dati Aci dei mezzi intestati agli utenti morosi e direttamente agganciati alle singole posizione all'interno del gestionale della riscossione.

La fase successiva per chi usa il gestionale è quella di verificare se effettivamente il veicolo sia di proprietà del debitore e se non sussistono cause ostative all'iscrizione del gravame, in tal caso il preavviso di fermo potrà essere notificato.

Nel preavviso di fermo ci sarà una clausola che comunica che in caso di mancato pagamento si procederà con l' iscrizione del fermo amministrativo.

Per la stampa e spedizione stessa procedura utilizzata per l'emissione delle ingiunzioni.

I preavvisi di fermo non pagati nei termini di legge (30 gg dalla notifica) si trasformano in fermi amministrativi.

I fermi iscritti saranno comunicati agli interessati tramite avviso spedito con posta ordinaria e opportunamente allegati nella banca dati del gestionale.

Nel caso il fermo fosse pagato, il personale che utilizza il gestionale spedirà l'atto di revoca a mezzo raccomandata al contribuente interessato per cancellare l'iscrizione del fermo amministrativo, anche questo atto verrà archiviato nel database.

Se il contribuente non fosse proprietario di alcun mezzo occorrerà procedere con eventuale azione esecutiva i pignoramenti.

Il pignoramento è il primo atto con cui si avvia l'esecuzione forzata, laddove non è stato possibile recuperare il credito con le azione cautelative.

L'azienda di riscossione utilizza alcune forme di pignoramento il pignoramento presso terzi e il pignoramento mobiliare.

Il Pignoramento presso terzi viene utilizzato quando il debitore percepisce uno stipendio o è titolare di un c/c bancario.

L’azienda di riscossione inoltra l’elenco dei potenziali esecutandi a delle Società di servizi che sotto pagamento, eseguono una serie di verifiche e indagini utili all’individuazione del terzo (datore di lavoro o c/c bancario).

Ottenuta l’informazione, gli ufficiali della riscossione di sede predisporranno l’atto di pignoramento che verrà inviato sia al debitore che al terzo (datore di lavoro o banca) .

Se invece non è stata reperita nessuna informazione di un possibile terzo tenterà la via del pignoramento mobiliare.

Il pignoramento mobiliare è un’azione forzata che viene eseguita sul territorio da parte dell’ufficiale di riscossione della società M.T. (se è competente territorialmente) oppure dall’ufficiale giudiziario del tribunale di competenza, previo invio dell’intimazione di pagamento (precetto) utile alla riapertura dei termini nel caso in cui gli atti ingiuntivi sottesi siano divenuti inefficaci.

Gli atti vengono predisposti manualmente dai nostri ufficiali della Riscossione ma archiviati digitalmente dentro gli archivi del gestionale.

Questo spiegato fino ad ora è la procedura che si esegue nella coattiva quando un contribuente non paga una o più posizioni.

Il software che è stato sviluppato da zero è un web service che tramite delle richieste Soap ottiene alcuni dati dal gestionale della riscossione, descritto qui sopra e li restituisce ad un portale.

Il portale è stato progettato da terzi secondo specifiche richieste dall’azienda e dovrà avere una determinata forma e struttura.

1.4 PORTALE COATTIVA

Il portale della coattiva risponde ed è visibile all'indirizzo :
https://www.ingiunzionifiscali.it/.

Il sito mostra i dati per i comuni in concessione, un comune in concessione significa che l’azienda di riscossione agisce per nome proprio e ha come cliente il comune.
Non tutti i comuni sono in concessione alcuni sono in affidamento, cioè agiamo per conto del comune sotto il nome del comune facendo il servizio di riscossione.
I comuni in affidamento hanno lo stesso portale, con le stesse funzionalità ma con grafica e riferimenti diversi.

Il portale della coattiva di un comune in affidamento risponde ed è visibile all'indirizzo
https://www.ingiunzionifiscali.it/NomeDelComune/.

Il sito del portale deve essere abilitato in dei determinati parametri nel gestionale come si vede in figura 1.1

Figura 1.1 : parametri per abilitare l'uso del portale

Il comune una volta attivato spedirà atti forniti di un riquadro che indica, come accedere al portale (figura 1.2).

Il QRCODE sarà presente solo sui nuovi atti che verranno generati dopo che si è abilitato, per un determinato comune l'accesso al portale.

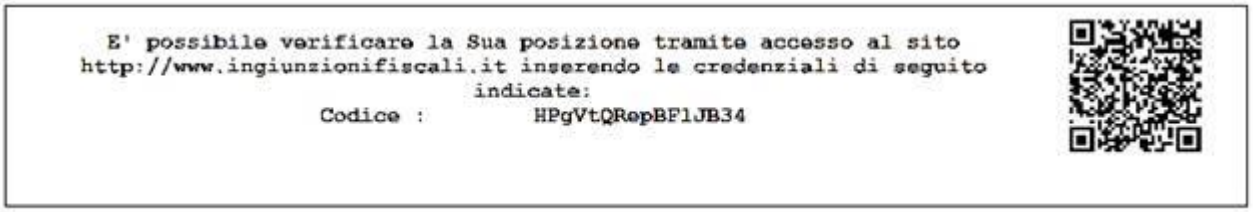


Figura 1.2: riquadro che indica al contribuente come usare il codice

Nel gestionale della riscossione sarà visibile il codice e l'ultima data di accesso al portale del soggetto come possiamo vedere in figura 1.3.

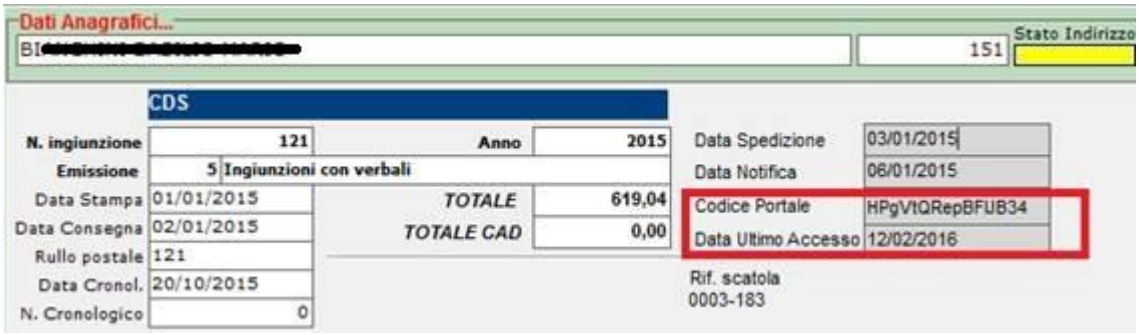


Figura 1.3: visualizzazione codice portale e data ultimo login nel gestionale

In figura 1.4 viene visualizzata la pagina di login del portale, utilizzando il codice contenuto nell'atto si effettua il login.

Figura 1.4: pagina login portale

Il comune in affidamento possiede una grafica del portale diversa dalla classica (figura 1.5).

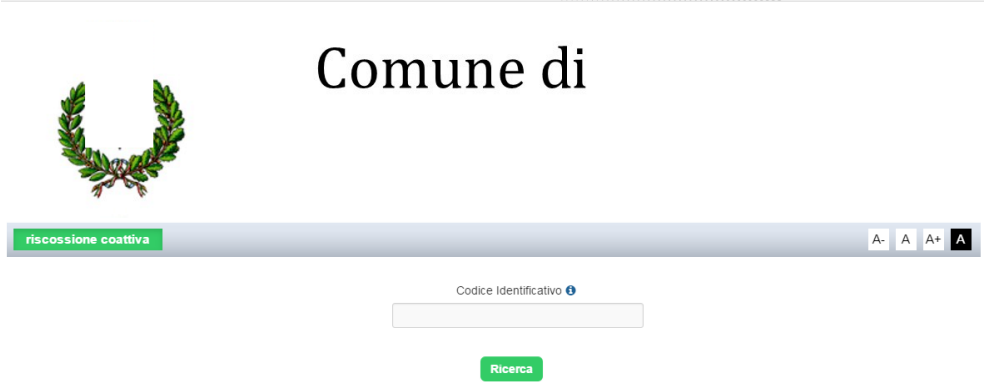


Figura 1.5: pagina login portale comune in affidamento

In alto nella prima parte, vi è il nome del comune in affidamento e a lato il simbolo del comune.

Come già avevo scritto prima i portali sono e devono essere identici devono cambiare solo i riferimenti di chi esegue la riscossione, questo perché il contribuente deve sapere con precisione chi sta eseguendo la parte della riscossione dei crediti.

Il portale fa visualizzare l'ultimo atto inviato, quindi se avete ricevuto un ingiunzione, un sollecito o un preavviso e accedete al portale con il codice di un atto precedente, il portale farà riferimento sempre all'ultimo atto inviato(ovviamente saranno disponibili anche i riferimenti degli atti precedenti all'ultimo inviato).

1.4.1 RIEPILOGO SITUAZIONE COATTIVA

Nella prima pagina vi è un riepilogo con le indicazioni del comune di riferimento(a cui non si sono pagati i verbali o le fatture/accertamenti), i dati anagrafici del soggetto coinvolto, il dettaglio dell'atto con gli importi totali, la data di notifica dell'atto, ovvero quando il postino ha dato l'atto in mano al destinatario, questo dato però è visibile se e solo se è presente la data di notifica nel nostro gestionale, perché può capitare che in caso le poste non ci abbiamo comunicato tempestivamente la data di notifica oppure vi è stata una notifica CAN, questa data non sia ancora disponibile nel nostro gestionale.
Quindi la data di notifica è disponibile solo quando poste ci comunica la data di avvenuta consegna tramite il postino.
Inoltre è presente lo stato dell'atto:
ovvero una breve informazione che indicherà la posizione di quell'atto, per esempio se è stato pagato oppure che tipo di notifica a ricevuto oppure se è stato presente un discarico o un ricorso per quella posizione;
ed una descrizione per il metodo di pagamento cioè i vari riferimenti a chi effettuare il pagamento (esempio conto corrente bancario. numero e a chi è intestato oppure conto corrente postale).
Infine vi è lo storico ingiunzione, questa parte indica tutto il procedimento che è stato effettuato dall'agenzia di riscossione da quando abbiamo ricevute le posizione non pagate, fino ad oggi;
in questa sezione ci sono i riferimenti agli atti precedenti(se presenti), il numero identificativo dell'atto, quando sono stati notificati, se risultano scaduti e non pagati oppure se risultano pagati.

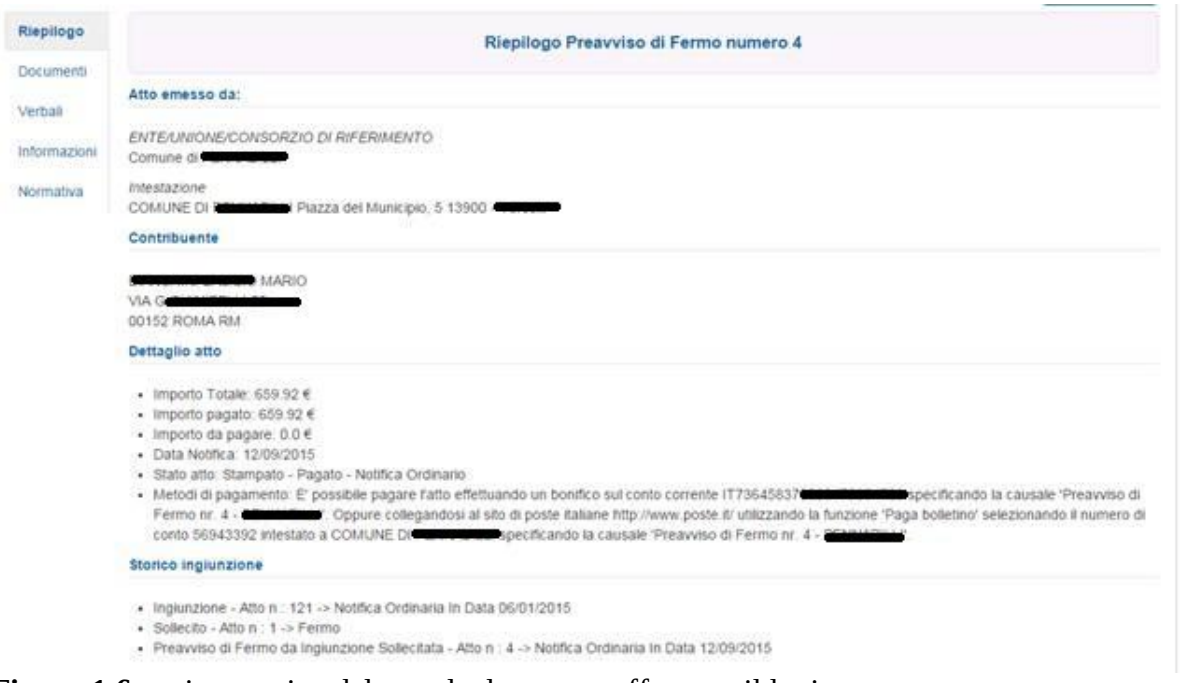


Figura 1.6 : prima pagina del portale dopo aver effettuato il login

1.4.2 DOCUMENTI

nella sezione documenti potremo scaricare le copie degli atti e delle notifiche di essi (ovviamente questi dati si visualizzano se sono stati archiviati nel nostro gestionale).
Le copie dell'atto, sono gli atti che sono stati recapitati a casa del contribuente e che sono stati digitalizzati nel nostro gestionale, sottoforma di PDF.
Invece la copia della notifica è l'immagine che poste ci ha comunicato, dove vi è l'immagine della scansione della cartolina verde con la firma di chi ha ricevuto l'atto dal postino, con il tipo di notifica e con tutte le informazioni riguardante la notifica.

Figura 1.7: elenco allegati disponibili

1.4.3 VERBALI/FATTURE/ACCERTAMENTI

Questa sezione mostra l'elenco delle posizioni (figura 1.8) che non erano state pagate al comune e che ora sono passate, nella fase della coattiva tramutandosi in un atto della coattiva,creato dal programma gestionale e che contiene i riferimenti ad esse.

Nel caso per, la nostra attività di riscossione dei crediti, riguardi le multe del codice della strada e che tali multe siano state spedite in precedenza da un nostro altro programma dell'azienda.
Avremmo disponibili nel nostro gestionale e quindi che facciamo vedere al contribuente attraverso il nostro portale, una copia digitale (standard per l'azienda) che descrive nel dettaglio i verbali ricevuti.
Dettagli che riguardano:

- chi ha ricevuto la multa
- per cosa si è stati multati
- il numero identificativo della multa(verbale)
- riferimenti anagrafici di chi ha preso la multa
- Dati del veicolo che ha ricevuto la multa
- Il tipo e quando è stata ricevuta la notifica della multa

Se questi oltre ai dati che elencano le posizioni non pagate, sono presenti nel nostro gestionale anche questi allegati, allora diamo la possibilità di visionarli, mettendo a disposizione una comoda griglia che elenca tutti i verbali e che per ogni verbale(se presente) dia la possibilità di scaricare l'allegato con le informazioni di esso

Figura 1.8: elenco fatture/accertamenti/verbali

Cliccando su visualizza, pulsante che non è visibile, nel caso non vi sia un allegato disponibile, vedremo il file che descrive dettagliatamente il verbale.

Generale			
Infrazione Nr.	V 500581571/2011 (*)	**Protocollo Nr.**	24/2011
Accertato il	01/01/2011 12:00	**Inserito il**	16/02/2011 Lucarini [redacted]

Stato verbale	
Stato	Notificato

Veicolo					
Tipo	AUTOVEICOLO	**Targa**	AA789AA	Italiana	
Marca		**Telaio**			

Localizzazione
VIA San Vitale sdsdsd d 12 (Barrali)

Accertatori	
1°	Mat. 1 [redacted]

Violazione/i	
1°	S [redacted] CIRCOLAVA IN C.A. NONOSTANTE SOSPENSIONE DELLA CIRCOLAZIONE DISPOSTA CON ORDINANZA DEL SINDACO PER MOTIVI DI SICUREZZA
Mot. Mancata Cont.	Accertamento della violazione in assenza del trasgressore.
Sanz. Accessorie	Sospensione della patente di guida

Anagrafiche	
Obbligato	
[redacted] MARIO	
Nato a [redacted] il 14/08/1949	
Res. ROMA(RM)in [redacted] 68	
Patente N. RM1093245 Rilasciata il 31/12/1985	

Importi					
Intero	80,00	**Ruolo**	159,00	**Ingiunzione**	0,00
Spese Notif.	10,80	**Spese varie**	0,00	**Spese Totali**	10,80
Pagato	0,00	**Dovuto a oggi**	169,80		

Date					
1° Notifica	23/12/2010(Can)	**2° Notifica**		**3° Notifica**	
1° Notifica Ing.		**2° Notifica Ing.**		**Ruolo**	
Ricerca Dati		**Riapertura Ter.**		**Stampa**	
Ricorso		**Invio Ricorso**		**Archiviazione**	
Pagamento					

Figura 1.9: esempio allegato verbale

1.4.4 INFORMAZIONI

La sezione informazioni indica la sede legale e operativa di chi esegue la riscossione con vari dettagli (telefono, mail pec, indirizzo delle sedi), queste informazioni sono presenti in ogni atto stampato e mandato al contribuente, che non ha regolarizzato le sue posizioni.
Sono informazioni molto importanti da fare vedere, perché il contribuente sapere a chi fare riferimento nel caso, voglia contattare chi gestisce la riscossione per:

- Informazioni generali

- Richiesta di rateizzazioni
- Informazioni per quanto riguarda la possibilità di fare ricorso o il diritto ad avere un discarico della posizione per vari motivi legali.

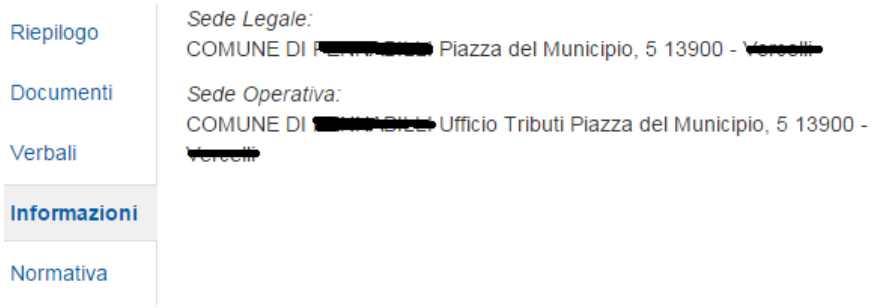


Figura 1.10 sezione informazioni portale

1.4.5 NORMATIVA

Vi è una sezione in cui vi sono i riferimenti alla normativa, per cui si spiega che l'agenzia di riscossione agisce a norma di legge e che fornisce tutte le informazioni su ciò che non si è pagato e su quello che si rischia nel caso, si continui a non regolarizzare le proprie posizioni.

Figura 1.11: pagina contenente riferimenti alla normativa

Il web service, interfaccerà il gestionale della riscossione con il portale creato da terzi, lo scopo di questo progetto è dare visibilità ad alcuni dati (sempre nel rispetto della privacy), in modo che il contribuente abbia una visione più chiara delle sue posizioni aperte.
I dati richiesti riguardano :

- Le informazioni anagrafiche del contribuente moroso
- Le informazioni sull'ultimo atto relativo alla propria storia coattiva
- La storia che riguarda tutti gli atti che il contribuente ha ricevuto, relativi a verbali/fatture/accertamenti che non ha pagato
- Le informazioni dell'ente che ha emesso gli atti della coattiva
- La possibilità di avere le copie digitali di quello che si è ricevuto per posta

- Indicazioni sulle modalità di pagamento
- Informazioni normative sulla procedura della coattiva

Il gestionale funziona in modo parametrizzato e quindi diverso, per ogni cliente/comune/ente che richiede il servizio di riscossione all'azienda, ogni singolo dettaglio è parametrizzabile quindi rende unico ogni cliente.
Per esempio ogni cliente avrà quindi diverse modalità di pagamento da un altro e ovviamente diversi riferimenti bancari/postai per la regolarizzazione della posizione.
Il web service dovrà funzionare in maniera analoga a quello che fa vedere il gestionale e saper filtrare tra i dati per ottenere quello giusto, senza che un contribuente veda informazioni sbagliate o non sue, attraverso il portale.
Questo servizio è estremamente necessario ai giorni nostri in cui ogni cosa è digitalizzata e resa disponibile sul web.

2 ANALISI DEL WEB SERVICE

Definito cosa fanno gli strumenti già esistenti nell'azienda, parliamo dei motivi per cui si è voluto creare un web service, che fornisse informazioni ai contribuenti.
Durante l'esplosione di internet, negli anni novanta, le aziende hanno mostrato verso questo nuovo strumento un interesse particolare.
In particolare, dopo aver visto una forte diffusione dei sistemi informativi all'interno delle aziende, si è sentita la necessità di poter integrare piattaforme differenti.
Questo tipo di esigenza sorge non solo tra aziende, ma addirittura all'interno della medesima azienda nella quale, per motivi storici e organizzativi, diverse divisioni hanno operato scelte tecnologicamente differenti nella realizzazione del proprio sistema informativo;
Questa situazione è molto diffusa nelle grandi aziende, specie se al loro interno ci sono altre aziende più piccole (magari acquisite), che devono per necessità continuare ad usare i loro sistemi informativi, ma che devono uniformarsi anche con la realtà aziendale esistente.
Da qui nasce l'esigenze di avere uno strumento che dia la possibilità a realtà diverse di interfacciarsi in maniera unica al sistema informativo che fornisce i dati.
In più questo strumento non serve solo internamente ma principalmente per fare in modo che ogni azienda possa fornire servizi a clienti esterni (magari facendo pagare il servizio).
C'è però un problema, di vista tecnologico affinché questa interoperabilità sia fattibile, le aziende devono accordarsi su un linguaggio comune di descrizione dei servizi in modo tale da riconoscere cosa un sistema mette a disposizione, solitamente in questa azienda il linguaggio utilizzato è Java.
La soluzione tecnologica adatta alla risoluzione del problema aziendale di interfacciare, i propri dati con sistemi esterni, viene risolta dai Web Services, che permettono di usare il web come canale di trasmissione per dare la possibilità di fornire dati opportunamente filtrati e lavorati.

Per la realizzazione di questo strumento l'azienda ha richiesto che venisse sviluppato utilizzando i seguenti framework, linguaggi e strumenti:

2.1 APACHE CXF

Apache CXF è un framework sviluppato e mantenuto dalla fondazione Apache.
Il suo nome deriva dalla fusione di due progetti (Celtix e Xfire) e l'obiettivo è quello di fornire delle astrazioni e degli strumenti di sviluppo per esporre varie tipologie di servizi web.
È basata sul framework Spring (IoC, AOP, Web MVC, Test)
APACHE CXF permette a diversi applicativi Java (vedi elenco sopra) di interfacciarsi al web service CXF.
(spiegazione dell'immagine 2.1 la foglia rappresenta il framework Spring che contiene CXF e da la possibilità di interfacciarsi, con tutti gli applicativi intorno a lui app, cluod data ecc).

Figura 2.1: come si interfaccia CXF con il mondo estrno

I punti di forza di CXF sono :

- Supporta entrambe le categorie di WS moderni: SOAP e REST
- Permette approcci Code First e Contract First
- Generazione di WSDL (o WADL) lato server
- Generazione del codice per il client e il server

Quindi permette di creare velocemente un web service, sia Soap che Rest, con relativa creazione automatica del WSDL.
Si tratta di uno strumento perfetto per la nostra esigenza aziendale, perché unisce uno strumento semplice, veloce e potente;
che si può integrare perfettamente con tutte le realtà aziendali (anche se in questo caso, per il momento, si interfaccerà solamente al portale che rende visibile i dati).
Un documento WSDL(Web Services Description Language) descrive un web service.
In questo documento si specifica la posizione del servizio, e le modalità del servizio, utilizzando questi elementi principali:
<definitions> <types>definisce il data type</types>

<message> definisce come I dati vengono comunicati</message>

<portType> imposta le operazioni.</portType>

<binding>definisce le specifiche del protocollo e del formato </binding> </definitions>

Come è strutturato CXF:
Basata su XML:

- Configurazione base su file XML

Basata su due file xml che definiscono le impostazioni ecco un esempio:

- Web.xml

```
<context-param>
<param-name>contextConfigLocation</param-name>
<param-value>WEB-INF/beans.xml</param-value>
</context-param>
<listener>
<listener-class>o.s.w.c.ContextLoaderListener</listenerclass>
</listener>
<servlet>
<display-name>CXF Servlet</display-name>
<servlet-name>CXFServlet</servlet-name>
<servletclass>o.a.cxf.transport.servlet.CXFServlet</servlet-class>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>CXFServlet</servlet-name>
<url-pattern>/*</url-pattern>
</servlet-mapping>
```

- Beans.xml

```
<import resource="classpath:META-INF/cxf/cxf.xml" />
<import resource="classpath:META-INF/cxf/cxf-servlet.xml" />
<context:annotation-config />
<context:component-scan basepackage="
```

```
it.maggioli.informatica.wsmaggioliservice" />
<bean id="serviceImpl" name="defaultImpl"
class="i.m.i.w.WsMaggioliServiceImpl" />
<jaxws:endpoint id="WsMaggioliService"
implementor="#serviceImpl"
implementorClass="……." address="/WsMaggioliService">
<jaxws:properties>
<entry key="mtom-enabled" value="true" />
</jaxws:properties>
<jaxws:inInterceptors>
<ref bean="wss4jInConfiguration" />
</jaxws:inInterceptors>
</jaxws:endpoint>
```

E sulle Annotazioni:

- Routing / funzioni con annotazioni sui metodi

Le Annotazioni sono un modo per aggiungere metadati nel codice sorgente Java che possono essere disponibili al programmatore durante l'esecuzione.

La dichiarazione di un'annotazione è una variante dei tag che sono stati aggiunti, in passato, per commentare le sezioni.

Le annotazioni prendono la forma di una dichiarazione di interfaccia con un @ che le precede e opzionalmente marcate con una meta-annotazione.

Ecco un esempio @XmlElement(required = **true**).

Per questo progetto verrano utilizzate richieste SOAP.

SOAP E' l'acronimo di Simple Object Access Protocol e rappresenta uno standard per lo scambio di messaggi tra componenti distribuiti.

Può operare su diversi protocolli, ma il più utilizzato è quello HTTP.

I messaggi sono in formato XML e seguono la configurazione Head – Body.

L'Header è opzionale e contiene informazioni circa il routing, la sicurezza e le transazioni, mentre il Body è obbligatorio e contiene il messaggio vero e proprio scambiato tra le due parti (chiamato in gergo Payload).

2.2 SPRING FRAMEWORK

Framework Java molto potente e molto diffuso, è riconosciuto anche da importanti vendor commerciali quale framework di importanza strategica.

La prima versione venne scritta da Rod Johnson e distribuita con la pubblicazione del proprio libro "*Expert One-on-One Java EE Design and Development*" (Wrox Press, ottobre 2002).

All'inizio il framework venne distribuito sotto Licenza Apache nel giugno 2003.

La prima pubblicazione importante è stato l'1.0 del marzo 2004, seguito da due successive distribuzioni importanti nel settembre 2004 e nel marzo 2005.

Spring è stato largamente riconosciuto all'interno della comunità Java quale valida alternativa al modello basato su Enterprise JavaBean (EJB).

Rispetto a quest'ultimo, il framework Spring lascia una maggiore libertà al programmatore fornendo allo stesso tempo un'ampia e ben documentata gamma di soluzioni semplici adatte alle problematiche più comuni.

Sebbene le peculiarità basilari di Spring possano essere adottate in qualsiasi applicazione Java, esistono numerose estensioni per la costruzione di applicazioni *web-based*(applicazioni Web) costruite sul modello della piattaforma Java EE.

L'utilizzo di Spring in questo progetto ci ha permesso di implementare queste librerie fondamentali:

- Manipolazione dati
  - Hibernate ORM / OGM
  - JPA
- Web Services
  - CXF
- Views
  - Freemarker
  - Velocity
  - Tiles
  - Thymeleaf

Utilizza inoltre questi Componenti Base:

- Contenitore IoC
  - Aspect Oriented Programming
  - Testing
- Accesso Dati
  - Transazioni
  - JDBC
  - ORM
  - XML
- Web
  - MVC Framework
  - Portlet
  - WebSocket

E questa importante caratteristica:
Aspect Oriented Programming

- Paradigma di programmazione per la separazione di interessi che incrociano più parti (separation of crosscutting concerns)
- Inoltre in Spring:
  – Approcci basati su XML o Annotazioni
  – Normalmente basato su proxy dinamici, richiede la combinazione interfaccia + implementazione
  – Possibilità di proxy su classi o di usare il weaving di AspectJ

– Supporto di buona parte delle funzionalità di AspectJ: Introduzioni, modelli di istanziazione.

Inoltre per comprendere a fondo le potenzialità del framework Spring bisogna prima introdurre i concetti di Inversion of Control (IoC) e Dependency Injection (DI).

L'Inversion of Control è un principio architetturale nato basato sul concetto di invertire il controllo del flusso di sistema (Control Flow) rispetto alla programmazione tradizionale.

Questo principio è molto utilizzato nei framework e ne rappresenta una delle caratteristiche basilari che li distingue dalle API.

Nella programmazione tradizionale la logica di tale flusso è definita esplicitamente dallo sviluppatore, che si occupa tra le altre cose di tutte le operazioni di creazione, inizializzazione ed invocazione dei metodi degli oggetti.

Inversion of Control invece inverte questo control flow facendo per fare in modo che non sia più lo sviluppatore a doversi preoccupare di questi aspetti, ma direttamente il framework, che reagiràse occupandosene per suo conto.

Il termine Dependency Injection (DI) invece,si riferisce ad una specifica implementazione dello IoC rivolta ad invertire il processo di risoluzione delle dipendenze, facendo in modo che queste vengano iniettate dall'esterno.Banalmente, nel caso della programmazione orientata agli oggetti, una classe_1 si dice dipendente dalla classe_2 se ne usa in qualche punto i servizi offerti (vedi Figura 2.2).

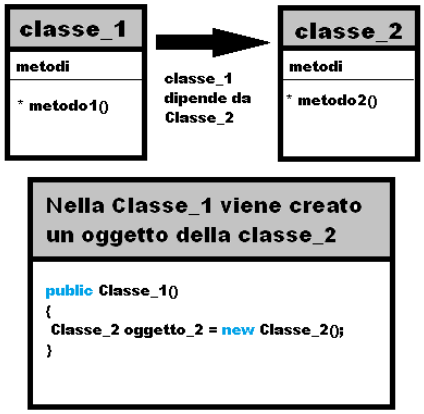


Figura 2.2: esempio di dipendenze

Perché questo tipo di collaborazione abbia luogo la classe_1 ha diverse alternative:

istanziare e inizializzare (attraverso costruttore o metodi setter) la classe_2, ottenere un'istanza di B attraverso una factory oppure effettuare un lookup attraverso un servizio di naming (es JNDI).

Ognuno di questi casi implica che nel codice della classe_1 ci sia la logica di risoluzione della dipendenza verso la classe_2.

Esempi implementazione IoC.

● Xml

```
<?xml version="1.0" encoding="UTF-
8"?>
<beans>
        <bean id="..." class="...">
```

```
        <!-- collaborators and
        configuration for this bean go here -->
        </bean>
        <bean id="..." class="...">
        <!-- collaborators and
        configuration for this bean go here -->
        </bean>
<!-- more bean definitions go here
-->
</beans>
```

● Annotazioni

```
<context:annotation-config/>
@Service("myMovieLister")
public class SimpleMovieLister {
@Autowired
public void
setMovieFinder( MovieFinder
movieFinder) {
this.movieFinder = movieFinder; }
// ...}
@Repository
public class MovieFinderImpl
implements MovieFinder {
// ...}
```

● Java

```
@Configuration
@ComponentScan(baseP
ackages = "com.acme")
public class AppConfig {
@Bean
public MyService
myService() {
return new
MyServiceImpl();
} }
```

2.3 HIBERNATE

In informatica Hibernate è una piattaforma middleware open source per lo sviluppo di applicazioni Java, attraverso l'appoggio al relativo framework, che fornisce un servizio di Object-relational mapping (ORM) ovvero gestisce la persistenza dei dati sul database attraverso la rappresentazione e il mantenimento su database relazionale di un sistema di oggetti Java.

Come tale dunque, nell'ambito dello sviluppo di applicazioni web, tale strato software si frappone tra il livello logico di business o di elaborazione e quello di persistenza dei dati sul database (Data Access Layer).

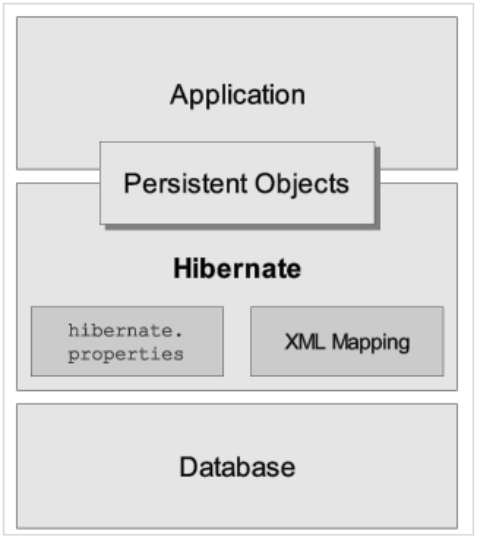


Figura 2.3: rappresentazione di Hibernate

Hibernate è stato originariamente sviluppato da un team internazionale di programmatori volontari coordinati da Gavin King; in seguito il progetto è stato proseguito sotto l'egida di JBoss, che ne ha curato la standardizzazione rispetto alle specifiche Java EE.Hibernate è distribuito in licenza LGPL sotto forma di librerie software da linkare nel progetto di sviluppo software.

Lo scopo principale di Hibernate è quello di fornire un mapping delle classi Java in tabelle di un database relazionale;
sulla base di questo mapping Hibernate gestisce il salvataggio degli oggetti di tali classi su database (tipicamente attributi di oggetti per ciascun campo dati della tabella).

Si occupa inoltre al rovescio del reperimento degli oggetti dal database, producendo ed eseguendo automaticamente le query SQL necessarie al recupero delle informazioni e la successiva reistanziazione dell'oggetto precedentemente "ibernato" (mappato su database).

L'obiettivo di Hibernate è quello di esonerare lo sviluppatore dall'intero lavoro relativo alla persistenza dei dati. Hibernate si adatta al processo di sviluppo del programmatore, sia se si parte da zero sia se da un database già esistente.

Hibernate genera le chiamate SQL e solleva lo sviluppatore dal lavoro di recupero manuale dei dati e dalla loro conversione, mantenendo l'applicazione portabile in tutti i database SQL. Hibernate fornisce una persistenza trasparente per Plain Old Java Object (POJO);
l'unica grossa richiesta per la persistenza di una classe è la presenza di un costruttore senza argomenti. In alcuni casi si richiede un'attenzione speciale per i metodi equals() e hashcode().

Hibernate è tipicamente usato sia in applicazioni Swing che Java EE facenti uso di servlet o EJB di tipo session beans.

2.3.1 OGGETTO / RELATIONAL MAPPING

Hibernate ORM consente agli sviluppatori di scrivere più facilmente le applicazioni di cui dati sopravvive il processo di applicazione.

Come un quadro Object / Relational Mapping (ORM), Hibernate si occupa di persistenza dei dati in quanto si applica ai database relazionali (tramite JDBC).

2.3.2 JPA PROVIDER

Oltre alla propria API "nativo", Hibernate è anche una implementazione della specifica Java Persistence API (JPA).

Come tale, essa può essere facilmente utilizzato in qualsiasi ambiente di supporto JPA incluse le applicazioni Java SE, application server Java EE, contenitori OSGi Enterprise, etc. la persistenza idiomatic Hibernate consente di sviluppare classi persistenti seguente idiomi orientati agli oggetti naturali, tra cui l'ereditarietà, polimorfismo, associazione, la composizione, e le collezioni framework Java.

Hibernate non richiede interfacce o classi di base per le classi persistenti e consente a qualsiasi struttura di classe o di dati per essere persistente.

Alte prestazioni Hibernate supporta l'inizializzazione differita, numerose strategie che vanno a prendere e il blocco ottimistico con delle versioni e il tempo automatico di timbratura.

Hibernate non richiede particolari tabelle del database o campi e genera gran parte del SQL in fase di inizializzazione del sistema invece che in fase di esecuzione.

Hibernate offre costantemente prestazioni superiori su codice JDBC direttamente, sia in termini di produttività degli sviluppatori e prestazioni di runtime.

Scalabilità Hibernate è stato progettato per lavorare in un cluster di server di applicazioni e fornire un'architettura altamente scalabile.

Affidabile Hibernate è ben noto per la sua eccellente stabilità e qualità, provata con l'accettazione e l'uso da decine di migliaia di sviluppatori Java.

Estensibilità Hibernate è altamente configurabile ed estensibile.

2.3.3 VERSIONI DI HIBERNATE

La versione 3 di Hibernate arricchisce la piattaforma con nuove caratteristiche come una nuova architettura Interceptor/Callback, filtri definiti dall'utente, e annotazione stile JDK 5.0 (Java's metadata feature).

Hibernate 3 è vicino anche alle specifiche di EJB 3.0 (nonostante sia stato terminato prima di EJB 3.0 le specifiche erano già state pubblicate dalla Java Community Process) ed è usato come spina dorsale per l'implementazione EJB 3.0 di JBoss.

Nel dicembre 2011 è uscita la versione 4.0, e a gennaio 2012 la versione 4.01. Nel mese di agosto 2013 è stata resa disponibile la versione 4.2.4.

L’ultima versione è quella utilizzata nel progetto.

Hibernate è una risorsa essenziale con cui possiamo sviluppare il nostro progetto, ci permette di frapporci tra il database del gestionale e la nostra applicazione(web service) definendo i Models che sono le classi che definiscono un oggetto che si interfaccia tramite hibernate al database.

Con hibernate abbiamo la possibilità di effettuare qualsiasi operazione, che si esegue solitamente direttamente sul database.

Per esempio possiamo eseguire oltre alle banali query, anche script sql in cui si eseguono delle update, delle insert nelle tabelle.

Tutto questo gestendo dividendo la parte del database con quella dell’applicazione dando utili possibilità ovvero:

- definire in una classe una tabella oppure una vista in maniera parziale(senza implementare tutte le colonne)
- manipolando il dato delle colonne (per esempio una colonna del database numerica, può essere moltiplicata per una costante)

2.4 MAVEN

Maven è un progetto open source, sviluppato dalla Apache, che permette di organizzare in modo molto efficiente un progetto java.

In informatica Maven è un software usato principalmente per la gestione di progetti Java e build automation. Per funzionalità è similare ad Apache Ant, ma basato su concetti differenti.
Può essere paragonato all’altro progetto più conosciuto della Apache, Ant, ma fornisce funzionalità più avanzate. I vantaggi principali di Maven sono i seguenti:

- standardizzazione della struttura di un progetto compilazione;
- test ed esportazione automatizzate;
- gestione e download automatico delle librerie necessarie al progetto;
- creazione automatica di un semplice sito di gestione del progetto contenente informazioni.

Può essere usato anche in progetti scritti in C#, Ruby, Scala e altri linguaggi.

Il progetto Maven è ospitato da Apache Software Foundation, dove faceva parte dell'ex progetto Jakarta.Maven usa un costrutto conosciuto come Project Object Model (POM);

un file XML che descrive le dipendenze fra il progetto e le varie versioni di librerie necessarie nonché le dipendenze fra di esse. In questo modo si separano le librerie dalla directory di progetto utilizzando questo file descrittivo per definirne le relazioni.

Maven effettua automaticamente il download di librerie Java e plug-in Maven dai vari repository definiti scaricandoli in locale o in un repository centralizzato lato sviluppo.

Questo permette di recuperare in modo uniforme i vari file JAR e di poter spostare il progetto indipendentemente da un ambiente all'altro avendo la sicurezza di utilizzare sempre le stesse versioni delle librerie.

Le principali funzionalità che maven rende disponibili sono i seguenti:

- **compiler**: che permette di compilare i file sorgenti;
- **deploy**: che permette di depositare il pacchetto generato nel repository remoto;
- **site**: che permette di generare la documentazione del progetto;
- **archetype**: che permette di generare la struttura di un progetto a partire da un template.
- **clean**: che permette di cancellare i compilati dal progetto;
- **install**: che permette di depositare il pacchetto generato nel repository locale;

Ogni funzionalità mette a disposizione degli obbiettivi predefiniti.

Ogni obbiettivo riceve in ingresso dei parametri specifici oppure facoltativi o obbligatori.

Inoltre permette di creare un progetto a partire da uno scheletro predefinito(template), il quale stabilisce la scheletro/struttura/template base delle cartelle e dei file che devono essere creati.

Per esempio alcuni dei principali template possono essere:

- maven-archetype-bundles;
- maven-archetype-archetype.

Un'altra cosa molto utile ed interessante di MAVEN, è che la prima volta che il progetto viene compilato, si scarica tutte le librerie dipendenti necessarie alla compilazione.

Questa fase è un po' lunga, ma le prestazioni non saranno sempre le stesse.

Questo però solo la prima volta che si compila il progetto, nelle successive compilazioni non ci sarà più bisogno di scaricare dal repository le librerie (a meno che non ne siano state aggiunte altre), rendendo quindi pratico e veloce la compilazione delle dependecy dalla seconda compilazione in poi.

Maven gestisce tutto in modo che ogni modulo contiene la sua cartella target con il risultato della compilazione.

Nel modulo Web troviamo il file war(il nostro webservice) pronto per il deploy.

Se lo apriamo possiamo notare come nella directory WEB-INF/lib ci siano le due librerie dipendenti, quella relativa al modulo java e quella relativa a log4j.

Infine per me questo innovativo software project management tool è un ottima scelta per me(anche se è stato deciso da chi ha scelto gli strumenti) per i seguenti due motivi:

1. Perchè ormai è diventato uno standard, si è migliorato e confermato nel tempo, ed aggiunge sicuramente un valore aggiunto al nostro progetto, in termini di sviluppo e pulizia.

2. Perchè non possiamo rimanere indietro rispetto a questa tecnologia, che secondo me troverà sempre più spazio, e quindi, prima si utilizza meglio è.

2.5 ORACLE

Il database a cui attingiamo i dati è Oracle, precisamente Oracle 10.

Il gestionale interno dell'azienda sfrutta questo eccezionale RDBMS (Relational DataBase Management System), Oracle si presta benissimo alla richiesta di gestione di molti dati e di protezione e sicurezza dei dati(specie quelli sensibili);

In esso sono contenuti tutti i dati relativi alla riscossione e anche i riferimenti a dove sono le copie digitali degli atti e a tutti gli altri allegati.

Detto questo il nostro web service può funzionare con qualsiasi altro RDBMS, l'importante è che si modifichino i parametri del collegamento verso un altro tipo di RDBMS e che tabelle, che verranno tramutati in classi (Models) siano presenti (alla stessa maniera) anche in quest'altro sistema .

NB. I dati presenti nel database e la loro struttura non possono e non devono cambiare, perché questo comprometterebbe l'integrità del gestionale. Il nostro web service dovrà agire in maniera sempre uguale nel tempo e in simbiosi con il database dei dati.

Oracle è uno tra i più famosi database management system (DBMS), cioè sistema di gestione di basi di dati, scritto in linguaggio C.

Questi database server sono macchine ottimizzate per ospitare i programmi che costituiscono il database reale e sulle quali girano solo il DBMS e il software ad esso collegato.
I database server si occupano di fornire i servizi di utilizzo del database ad altri programmi ed ad altre macchine secondo la modalità client server.
Il server ha il compito di memorizzare i dati, ricevere le richieste dei client ed elaborare le risposte appropriate. Essi sono concepiti, oltre che per la memorizzazione dei dati anche per fornire un accesso rapido ed efficace ad una pluralità di utenti contemporaneamente e garantire protezione sia dai guasti che dagli accessi indebiti.

Un server Oracle è rappresentato fondamentalmente da due strutture, il database e l'istanza. Con il termine database (d'ora in poi DB) si indicano i file fisici in cui sono memorizzati i dati, mentre per istanza si intende l'insieme delle aree di memoria e dei processi di background necessari ad accedere ai dati, ovvero al DB.

L'architettura del server è complessa: ogni area di memoria nell'istanza contiene dati che non sono contenuti in altre e i processi di background hanno compiti ben precisi, tutti diversi fra loro.

Ogni DB deve obbligatoriamente fare riferimento almeno ad un'istanza, è anche possibile avere più di un'istanza per database, ma nella nostra guida faremo sempre riferimento a database a singola istanza.

Ciascun DB consiste di strutture logiche di memorizzazione, per immagazzinare e gestire i dati, (tabelle, indici, etc.) e di strutture fisiche di memorizzazione che contengono le strutture logiche.

I servizi offerti dalle strutture logiche del server sono indipendenti dalle strutture fisiche che le contengono.

Questo perché le strutture logiche possano essere progettate nello stesso modo indipendentemente dall'hardware e dal sistema operativo impiegati.

Quindi sia che abbiamo un'installazione di server Oracle su un qualsiasi sistema operativo esistente non troveremmo alcuna differenza nella progettazione delle strutture logiche di memorizzazione.

2.6 LINGUAGGIO JAVA

Tutto il progetto sarà sviluppato in linguaggio Java.

Java nasce sostanzialmente per creare un linguaggio Object Oriented (orientato alle classi), che risolvesse principalmente due problemi presenti dell'epoca in cui fu creato(1995):

- essere orientato agli oggetti;
- essere indipendente dalla piattaforma;
- contenere strumenti e librerie per il networking;
- essere progettato per eseguire codice da sorgenti remote in modo sicuro.
- garantire maggiore semplicità rispetto al C++, per la scrittura e la gestione del codice
- permettere la realizzazione di programmi non legati ad una architettura precisa.

Il primo degli obbiettivi fu affrontato liberando il programmatore dall'onere della gestione della memoria (e togliendo dalle sue mani la gestione dei puntatori) creando il primo linguaggio destinato alla grande diffusione, basato su un sistema di gestione della memoria;

la soluzione fu il gargbage collection in cui automaticamente la memoria viene assegnata e rilasciata a seconda delle esigenze del programma, senza che il programmatore se ne debba curare (una vera rivoluzione).

Il secondo punto fu invece affrontato applicando il concetto di macchina virtuale, facendo sostanzialmente in modo che i programmi non fossero compilati in codice macchina (nativo) ma in una sorta di codice "intermedio" (chiamato bytecode) che non è destinato ad essere eseguito direttamente dall'hardware ma che deve essere, a sua volta, interpretato da un secondo programma, la macchina virtuale appunto.

Altre caratteristiche :

- **Orientamento agli Oggetti ,** l'orientamento agli oggetti, si riferisce a un moderno metodo di programmazione e progettazione. L'idea principale della programmazione ad oggetti consiste nel rendere il software la rappresentazione di entità reali o astratte ma ben definite (oggetti). Questi oggetti, come nella vita pratica hanno proprietà rappresentate da valori, e qualità o meglio metodi: ciò che sanno fare questi oggetti.
- **Indipendenza dalla piattaforma ,** l'indipendenza dalla piattaforma, significa che l'esecuzione di programmi scritti in Java deve avere un comportamento simile su hardware diverso. Si dovrebbe essere in grado di scrivere il programma una volta e farlo eseguire dovunque. Questo è possibile con la compilazione del codice di Java in un linguaggio intermedio bytecode, basato su istruzioni semplificate che ricalcano il linguaggio macchina.
- **Esecuzione sicura del codice remoto,** la piattaforma Java fu uno dei primi sistemi a fornire un largo supporto per l'esecuzione del codice da sorgenti remote. Una applet Java

è un particolare tipo di applicazione che può essere avviata all'interno del browser dell'utente, eseguendo codice scaricato da un server web remoto.

2.7 XML

Extensible Markup Language (XML) deriva da SGML (Standard Generalized Markup Language), come lo è anche HTML.

XML:

- È un mezzo con cui trasportiamo i dati
- È un linguaggio ed anche uno strumento con cui codifichiamo i dati
- è un sistema attraverso cui i dati sono rappresentati, trasmessi e ricevuti dai destinatari

Xml è un linguaggio che utilizza tag e può essere utilizzato per immagazzinare dati(in maniera ordinata) è molto utilizzato per i web service, come accennato prima ci sono vari file di configurazione di CXf in xml, il wsdl è un xml, con questo linguaggio, riusciamo a codificare e configurare tutto quello che ci serve per creare il nostro web service.

2.8 ECLIPSE

Per sviluppare il progetto è stato scelto Eclipse, un IDE perfetto per sviluppare in Java.

Eclipse è un ambiente di sviluppo integrato multi-linguaggio e multipiattaforma.

Può essere utilizzato per la produzione di software di vario genere, il software di questo progetto verrà sviluppato come introdotto prima in Java.

La piattaforma di sviluppo è incentrata sull'uso di plug-in e chiunque può sviluppare e modificare i vari plug-in.

Una caratteristica interessante di Eclipse è che a differenza di altri IDE, nella sua configurazione iniziale Eclipse prova a compilare automaticamente e continuamente tutto il codice che scriviamo, appena lo scriviamo.

L'interfaccia è di facile utilizzo ecco un immagine per avere chiaro a livello generale come si presenta un progetto gestito con Eclipse.

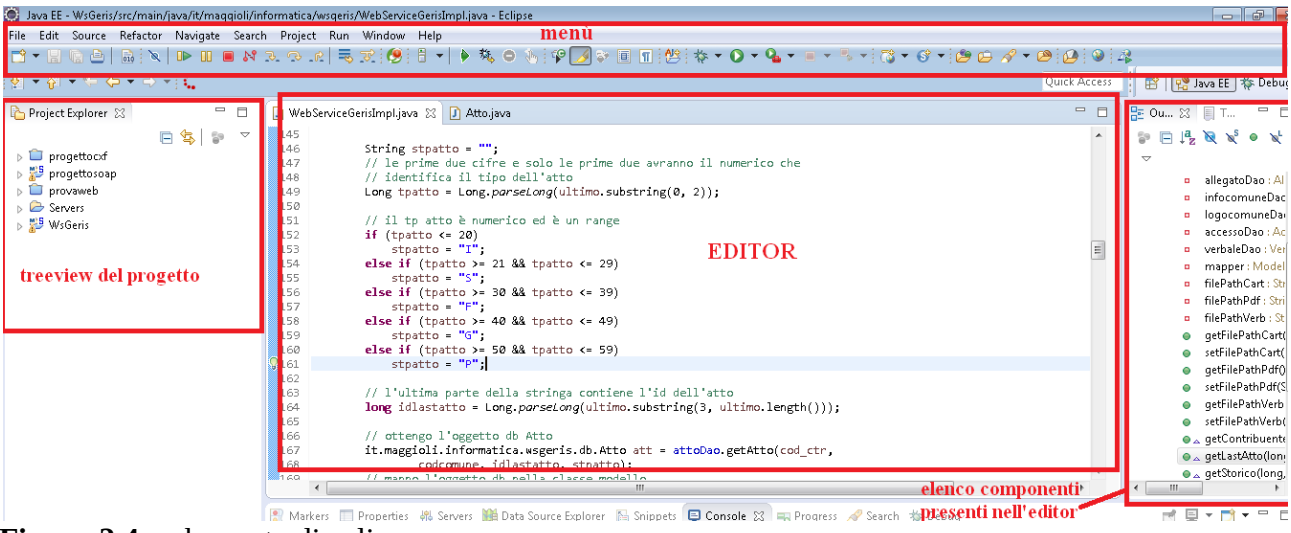


Figura 2.4: schermata di eclipse

Per i test del web service infine è stato utilizzato Soap UI è un programma per fare test sulle chiamate web service sia soap che rest, questo programma è semplice potente e gratis in più è anche open source. Una piattaforma di test perfetta per testare il nostro web service.

Come detto prima è molto semplice da usare, grazie all'interfaccia molto intuitiva, permette di creare in velocità e semplicità il programma di test.

Per farlo basta dargli in input il wsdl del nostro web service e in automatico ci mostrerà le richieste che possiamo fare indicandoci quali parametri inserire, la richiesta in input di un metodo viene fatta in xml e l'output ricevuto sarà sempre un xml, con la risposta o con l'errore in caso di problemi.

SoapUI permette anche diverse personalizzazione per configurare il proprio ambiente di test.

Ecco un immagine per spiegare il programma :

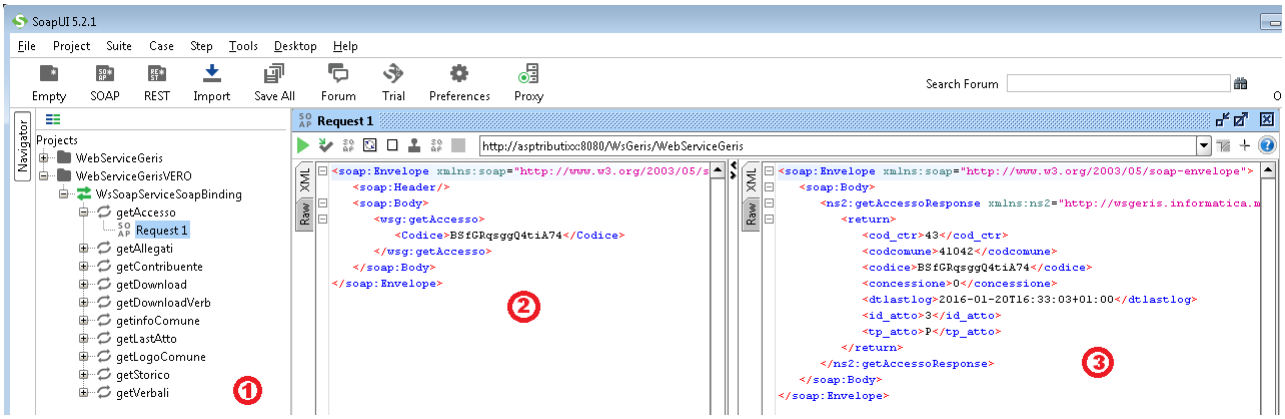


Figura 2.5: schermata del programma SOAP UI

Il punto 1 SoapUI elenca i metodi trovati nel wsdl del web service.

Nel punto 2 vi è l'xml con l'input di uno dei metodi scelti.

Nel punto 3 invece vi è l'output (la risposta), sempre in formato xml, del metodo richiamato nel punto 2 (input) con i parametri scritti dentro l'xml di input.

3 PROGETTAZZIONE E REALIZZIONE

Ora andiamo a trattare di come è stato affrontato il progetto, una volta capito l'obbiettivo e compreso meglio la situazione.

Abbiamo trattato prima di cosa già esiste e di cosa bisogna di realizzare, specificando poi in maniera particolareggiata, cosa siamo andati ad usare per realizzare questo web service.

Il web service dovrà interfacciare i dati del database del gestionale e per farlo dovranno essere già formattati, gestiti e pronti all'uso in modo che quando, ci sarà la richiesta (get) del metodo del web service, hibernate farà una query sugli oggetti definiti (come andremo a spiegare) in java e otterrà già il risultato che successivamente il web service darà in output a chi lo ha richiesto.

Andiamo ad analizzare come bisogna e come è stato sviluppato, il web service;
la struttura è la seguente visto che usiamo Hibernate ORM avremo:

- i metodi che appariranno nel web service, i quali risponderanno alle esigenze del portale
- i MODEL che sono le classi che rappresentano i dati nel risultato dei metodi
- il DAO dove ci saranno tutte le query Hibernate
- e le classi Database che sono le classi che saranno collegate alle tabelle/viste del database e in base a queste classi si potranno fare le query

Figura 3.1:struttura delle classi

Praticamente i passaggi di come funziona il web service sono:

il metodo viene chiamato da un software esterno, vengono passati i parametri richiesti, il metodo richiamerà il DAO che contiene i metodi che definiscono le query eseguite con HIBERNATE, la query potrà funzionare perché sono state definite le classi DATABASE che si collegano al database ORACLE del gestionale, quindi sarà come emulare una query SQL fatta direttamente su un database. Quindi il DAO restituirà un oggetto oppure una lista di oggetti di tipo classe DATABASE al nostro metodo, infine ci sarà un passaggio da classi DATABASE a quelle MODEL (perché le classi MODEL hanno delle logiche aggiuntive per gestire/modificare i dati del database), cioè i dati delle classi database verranno copiate in quelle dei MODEL, tutto questo poi viene restituito dal metodo (il suo output) al software che lo aveva richiesto.

Inoltre vi sono due file XML in cui ci sono le varie configurazioni del web service e sono:

- pom.xml contiene tutti i riferimenti alle dll, hai package a qualsiasi oggetto esterno dal nostro programma e che è stato implementato dentro il nostro progetto
- web.xml contiene i valori di eventuali variabili globali del progetto (vedi in futuro tratteremo delle variabili filePathCart, filePathPdf, filePathVerb), questo file serve inoltre per la configurazione di cxf

- beans.xml contiene tutte le configurazioni di cxf e di eventuali variabili globali, questo file è sempre necessario per cxf, deve essere inoltre creato nella maniera più corretta possibile, altrimenti il web service non funzionerà.
- context.xml contiene la configurazione per collegarsi al database del nostro gestionale, contiene tutte le informazioni per identificarlo e comunicare con esso.

```
<Context docBase="WsGeris" path="/WsGeris"
    reloadable="true">
    <Resource name="jdbc/WsGeris" auth="Container"
        factory="org.apache.tomcat.jdbc.pool.DataSourceFactory"
        localDataSource="true" type="javax.sql.DataSource" maxActive="30"
        maxIdle="30" maxWait="10000" username="[redacted]" password="[redacted]"
        driverClassName="oracle.jdbc.OracleDriver" url="[redacted]WORLD
        moveAbandoned="true" removeAbandonedTimeout="120" initialSize="3"
        validationQuery="select 1 from dual" testOnBorrow="true" testWhileIdle="true"
        timeBetweenEvictionRunsMillis="60000"
        connectionProperties="oracle.jdbc.timezoneAsRegion=false"/>/>
</Context>  <!-- driverClassName="oracle.jdbc.OracleDriver" url="jdbc:oracle:thin:@TRIBPROD3:1521:ORCL" -->
```

Figura 3.2: context.xml

Ora vediamo passo per passo come sono stati scritte tutte le parti del web service.

La prima parte che trattiamo della realizzazione sono i metodi del web service, i metodi sono contenuti in due file e sono:

- il primo che contiene le interfacce dei metodi, ovvero il WEBSERVICEGERIS (vedi figura 3.3 per un esempio), questo file descrive le firme dei metodi ed è possibile aggiungere, grazie alle annotazione dei controlli o delle definizioni, tutto senza scrivere righe di codice

```
@WebService
@BindingType(SOAPBinding.SOAP12HTTP_MTOM_BINDING)
@MTOM(enabled = true, threshold = 2048)
public interface WebServiceGeris {

    /**
     *
     * @param cod_ctr
     *            codice del contribuente
     * @param codcomune
     *            codice istat comune
     * @return tutte le informazioni anagrafiche del contribuente richiesto
     * @throws DatabaseException
     */
    // @WSDLDocumentation("A traditional form of greeting")
    Anagrafe getContribuente(
            @WebParam(name = "cod_ctr") @XmlElement(required = true, nillable = false) long cod_ctr,
            @WebParam(name = "codcomune") @XmlElement(required = true, nillable = false) long codcomune)
            throws DatabaseException;
```

Figura 3.3: interfaccia che verrà implementata dentro la classe del webservice

In questo esempio vediamo il metodo getContribuente, restituisce un oggetto di tipo Anagrafe e richiede due parametri (definiti come WebParam), parametri che non possono essere nulli e sono per forza richiesti, tutte le interfacce dei metodi definiti nel WEBSERVICEGERIS, sono definiti come l'immagine in figura 3.3.

- Nel WEBSERVICEGERISIMPL, invece ci sono i veri e propri metodi, la classe eredita l'interfaccia che abbiamo descritto prima

```
public class WebServiceGerisImpl implements WebServiceGeris {

    @Autowired
    private AnagrafeDao anagDao;// per i metodi dell'anagrafe

    @Autowired
    private UltimoAttoDao ultimoattoDao;// per i metodi per le informazioni
                                        // riguardo l'ultimo atto

    @Autowired
    private AttoDao attoDao;// per i metodi riguardanti il singolo atto

    @Autowired
    private StoricoDao storicoDao;// per i metodi riguardanti lo storico degli
                                  // atti

    @Autowired
    private AllegatoDao allegatoDao; // per i metodi riguardanti la lista degli
                                     // allegati

    @Autowired
    private InfoComuneDao infocomuneDao;// per i metodi riguardanti le
                                        // informazioni del comune

    @Autowired
    private LogoComuneDao logocomuneDao;// per ottenere il logo

    @Autowired
    private AccessoDao accessoDao;// per ottenere i dati dalla login

    @Autowired
    private VerbaleDao verbaleDao;// per ottenere tutti i verbali relativi ad
```

```
@Autowired
private ModelMapper mapper;// serve per "mappare" le classi db con dell
                           // classi oggetto a specchio

// path per le cartoline (costante)
private String filePathCart;// definite nel beans e nel web xml

// path per i pdf degli atti (costante)
private String filePathPdf;// definite nel beans e nel web xml

// path per gli xml dei verbali
private String filePathVerb;// definite nel beans e nel web xml

/* get e set delle costanti */
public String getFilePathCart() {
    return filePathCart;
}

public void setFilePathCart(String filePathCart) {
    this.filePathCart = filePathCart;
}

public String getFilePathPdf() {
    return filePathPdf;
}

public void setFilePathPdf(String filePathPdf) {
    this.filePathPdf = filePathPdf;
}

public String getFilePathVerb() {
    return filePathVerb;
```

Figura 3.4 e 3.5: la classe del web service

Dentro il WEBSERVICEGERISIMPL, possiamo dichiarare oltre hai metodi, un insieme di altre cose molto utili, per i metodi, si possono creare delle proprietà, delle variabili.

Per esempio come si vede nella figura qui di sopra, vengono dichiarate le variabili globali filePathCart, filePathPdf, filePathVerb avranno sempre un valore fisso definito nel file web.xml e servono per avere la radice del percorso di dove sono degli allegati che tratteremo più avanti.

Ora invece analizziamo ogni metodo in tutte le sue parti

3.1 GETACCESSO

Questo metodo controlla che il codice inserito durante il login, sia corretto e corrisponda ad uno dei codici che associa un atto.

Partiamo descrivendo la firma del metodo, il metodo è di tipo public in modo che chiunque possa vedere questo metodo, richiede un parametro di tipo string, che è il nostro codice che viene inserito nel momento del login; throws databaseexception è una classe(sempre da me creata) che solleva eccezioni di tipo database, nel caso di problemi con il risultato della query.

Il primo controllo che esegue è il controllo della lunghezza della stringa codice, se la stringa non è lunga 16 caratteri, viene sollevata un eccezione utilizzando la classe databaseexception.

```
public Accesso getAccesso(String codice) throws DatabaseException {

    // controllo la lunghezza per una sicurezza in più
    if (codice.length() != 16)
        throw new DatabaseException(
                "Error, codice/controcodice non conformi");

    char[] cod = codice.substring(0, 14).toCharArray();
    int resto = Integer.parseInt(codice.substring(14, 16));
    int chk = 0;
    for (int i = 0; i < cod.length; i++)
        chk += cod[i];

    if (chk % 91 != resto) {
        throw new DatabaseException(
                "Error, codice/controcodice non conformi");
    }

    // ottengo le informazioni per il relativo codice
    it.maggioli.informatica.wsgeris.db.Accesso acces = accessoDao
            .getInfoAccesso(codice);
    // se access è null vuol dire che c'è stato un problema nel dao quindi
    // le credenziali non sono corrette o
    // non esiste un codice nel db
    if (acces == null)
        throw new DatabaseException(
                "Error, credenziali sbagliate/inestistenti");

    // mappo l'oggetto db nella classe modello
    Accesso ac = mapper.map(acces, Accesso.class);
    return ac;
```

Figura 3.1.1: metodo getAccesso

Se la stringa è conforme vi è un ulteriore controllo prima di eseguire la query, questo controllo consiste nel prendere le ultime due cifre del codice (sono sempre numerici) e poi eseguire un ciclo

che somma il valore (ascii) di ogni carattere del codice dal primo carattere fino al 14°, se il numero chk in modulo 91 è diverso al numero contenuto nelle ultime due cifre del codice, il web service solleverà un'ulteriore eccezione.

Se il codice è tutto conforme si può procedere con la query, per eseguire la query ci avvaliamo di un metodo contenuto dentro una classe DAOIMPL (per ogni oggetto database vi è una classe DAOIMPL), ogni classe ha un'interfaccia che implementa i propri metodi, come vediamo nella figura 3.1.2.

```java
package it.maggioli.informatica.wsgeris.daos;

import it.maggioli.informatica.wsgeris.db.Accesso;

@Repository
@Transactional
public interface AccessoDao {

    Accesso getInfoAccesso(String codice);
}
```

Figura 3.1.2: interfaccia classe AccessoDao

Il metodo getInfoAccesso permette di eseguire la query dentro il database del gestionale (utilizzando una classe che definisce la tabella dentro il database), grazie ad una variabile di tipo sessionFactory. Questa particolare variabile permette di eseguire una query, che assegneremo ad un'altra variabile di tipo query, infine quest'ultima si possono impostare i parametri della query(gli attributi che hanno i : dentro la stringa della query), in questo caso codice trasmesso prima dal metodo del web service.

```java
public class AccessoDaoImpl implements AccessoDao {

    // serve x aprire la sessione x il db
    @Autowired
    private SessionFactory sessionFactory;

    @Override
    public Accesso getInfoAccesso(String codice) {

        // hql
        Query q = sessionFactory.getCurrentSession().createQuery(
                "from Accesso where codice = :codice");
        // i due punti sono i parametri si può usare anche ?
        q.setParameter("codice", codice);
        // q.setParameter("controCodice", controCodice);
        // inizializzo la variabile per il return
        Accesso a = null;

        // se q.unique result non da risultato genera un eccezzione
        try {
            a = (Accesso) q.uniqueResult();
        } catch (Exception ex) {

            return null;
        }
```

Figura 3.1.2: metodo getInfoAccesso

Eseguendo l'assegnazione del risultato della query al valore di ritorno del metodo dentro la classe DAOIMPL, potrebbe capitare che vi sia un errore o che la query non restituisca nessun valore, in quel caso restituiamo un eccezione altrimenti un oggetto di tipo Accesso che ora andiamo a vedere nel dettaglio dopo aver spiegato un'ultima parte di questo metodo.

```java
//definisce che è un entità
@Entity
// si da il nome della tabella/vista e lo schema
@Table(name = "TABACCESSIWEB", schema = "RISCOSSIONE")
public class Accesso {

    // definisco il nome di colonna
    @Column(name = "CODCOMUNE")
    // definisco il nome della attributo che rappresenta la colonna nel db
    private Long codcomune;

    @Column(name = "CODCTR")
    private Long cod_ctr;

    @Column(name = "IDRATTO")
    private Long id_atto;

    @Column(name = "TPATTO")
    private String tp_atto;

    @Id
    @Column(name = "CODICE")
    private String codice;

    @Column(name = "CONTROCODICE")
    private String concessione;
```

Figura 3.1.2: classe database Accesso

Dopo aver eseguito la query ottenendo un risultato positivo, eseguiamo uno script HQL, che esegue un update dentro la tabella, dove abbiamo eseguito la query, del gestionale modificando il campo che contiene una data e ora, che corrisponderà all'ultimo accesso dell'utente per quella posizione.

```java
        // eseguo una query di update per aggiornare la data di ultimo accesso
        Date dt = new Date();
        Query qupd = sessionFactory
                .getCurrentSession()
                .createQuery(
                        "Update Accesso set dtlastlog = :dtlastlog where codice = :codice  ");
        qupd.setParameter("dtlastlog", dt);
        qupd.setParameter("codice", codice);

        qupd.executeUpdate();

        return a;
    }
}
```

Figura 3.1.4: script update

L'oggetto che restituiamo è definito in una classe di tipo Database che vediamo nella seguente figura.

Questo è come si definisce una classe che descrive una tabella o una vista, dentro il database a cui ci colleghiamo.

L'annotation @TABLE con il parametro name descrive a quale tabella o vista ci interfacciamo, e con il parametro schema invece descrive, a quale schema risieda la tabella o vista. L'annotation Entity invece definisce che la classe è un'entità.

Dentro la classe (che dichiariamo sempre pubblica), vengono definiti i campi (con delle proprietà) che sono dentro la tabella/vista del database, il vantaggio più grande è che noi possiamo definire anche solo una parte dei campi, senza curarci degli altri, questa potenzialità è vitale perché in caso contrario avremmo dovuto definire ogni cosa della tabella/vista, con enorme spreco di tempo.

```java
// metodi get e set

public Long getCodcomune() {
    return codcomune;
}

public void setCodcomune(Long codcomune) {
    this.codcomune = codcomune;
}

public Long getCod_ctr() {
    return cod_ctr;
}

public void setCod_ctr(Long cod_ctr) {
    this.cod_ctr = cod_ctr;
}
```

Figura 3.1.5: come vengono dichiarate le proprietà in una classe

Le annotation @Column indicano che la proprietà si riferisce ad una determinata colonna dell'oggetto dentro il database, così facendo nella gestione del nostro web service possiamo, dare un nome diverso ad una colonna senza modificare niente dentro il database.

L'annotation @Id invece indica che il campo che stiamo definendo è chiave primaria.

NB ogni variabile di tipo private corrisponde ad una proprietà get e set che definisce la colonna dell'oggetto dentro il database.

Una volta ottenuto indietro il risultato della query, non rimane altro al metodo del web service che lavorarlo, per farlo ha bisogno di un mapper che faccia la conversione dell'oggetto definito della classe database, in uno di tipo Model (definito da me), che può essere identico a quello del database, ma potrebbe permettere la manipolazione di qualche dato nel caso se ne necessiti, prima di dare il risultato del metodo del web service.

La classe potete vederla, nella figura 3.1.6, si tratta come ho anticipato prima, di una semplice classe con gli stessi campi di quella del tipo Database, che potrebbe permettere una manipolazione del dato finale, nel caso ci sia il bisogno di farlo.

```
public class Accesso {

    private Long codcomune;

    private Long cod_ctr;

    private Long id_atto;

    private String tp_atto;

    private String codice;

    private String concessione;

    private Date dtlastlog;

    public Long getCodcomune() {
        return codcomune;
    }

    public void setCodcomune(Long codcomune) {
        this.codcomune = codcomune;
    }

    public Long getCod_ctr() {
        return cod_ctr;
    }

    public void setCod_ctr(Long cod_ctr) {
        this.cod_ctr = cod_ctr;
    }
```

Figura 3.1.6: classe model Accesso

3.2 GETINFOCOMUNE

Questo metodo, fornisce le varie informazioni del comune/ente a cui non si ha pagato il dovuto, questo metodo esegue due query, la prima ci fa ottenere l'identificativo del conto corrente a cui è legato l'atto. Mentre la seconda ci fornisce le informazioni che dobbiamo restituire.

```
@Override
public InfoComune getinfoComune(long idratto, long codcomune, String tp_atto)
        throws DatabaseException {

    long idccp = 1;
    // per sapere le informazioni del comune specifiche per l'atto in
    // questione cerco l'id del conto corrente di quell'atto
    idccp = attoDao.getIdccp(codcomune, idratto, tp_atto);

    // avendo il codice del comune e l'id del conto posso sapere le
    // informazioni dettagliate
    it.maggioli.informatica.wsgeris.db.InfoComune inf = infocomuneDao
            .getinfoComune(codcomune, idccp);

    // mappo l'oggetto db nella classe modello
    InfoComune infcom = new InfoComune(inf.getCodcomune(), inf.getComune(),
            inf.getId_ccp(), inf.getSede_legale(), inf.getSede_operativa(),
            inf.getIntestazione(), inf.getIntestazioneente());// mapper.map(inf,
                                                              // InfoComune.class);
    return infcom;

}
```

Figura 3.2.3: metodo getInfoComune web service

Inoltre in fondo come spiegato prima vi è la fase di "mappatura", della classe database con quella Model.

```java
//definisce che è un entità
@Entity
// si da il nome della tabella/vista e lo schema
@Table(name = "VWPARAMETRI4WEB", schema = "RISCOSSIONE")
// per gli id composti si deve creare una classe e implementare con questa
// annotation
@IdClass(InfoComunePK.class)
public class InfoComune {

    @Id
    // definisco il nome di colonna
    @Column(name = "CODCOMUNE")
    // definisco il nome della attributo che rappresenta la colonna nel db
    private Long codcomune;

    @Column(name = "COMUNE")
    private String comune;

    @Column(name = "ID_CCP")
    private Long id_ccp;

    @Column(name = "SEDE_LEGALE")
    private String sede_legale;
```

Figura 3.2.1: classe InfoComune

```java
@Repository
@Transactional
public class InfoComuneDaoImpl implements InfoComuneDao {

    @Autowired
    private SessionFactory sessionFactory;

    @Override
    public InfoComune getInfoComune(Long codcomune, Long id_ccp) {
        Query q = sessionFactory
                .getCurrentSession()
                .createQuery(
                        "from InfoComune where id_ccp = :id_ccp and codcomune = :codcomune");
        q.setParameter("id_ccp", id_ccp);
        q.setParameter("codcomune", codcomune);
        return (InfoComune) q.uniqueResult();
    }
```

Figura 3.2.2: GetInfoComune classe DAOIMPL

La query che viene eseguita nella classe DAOIMPL, è quella in figura 3.2.2, una volta ottenuti tutti i parametri si può eseguire la query sempre usano l'oggetto SessionFactory.

In questo caso la query restituirà sempre un unico risultato (questo è confermato perché i dati che diamo in pasto nella query sono sempre quelli del database e la logica del database non permette errori)

NB id_ccp lo otteniamo con un'altra query che discuteremo con il metodo GetLastAtto.

La classe database qui usata è InfoComune, questa classe oltre alle cose definite prima, ha un ulteriore Annotation, chiamata @IdClass che permette di definire una chiave composta.

Perché se nella tabella la chiave primaria fosse definita da una sola colonna, non bisogna fare nulla se non indicare quale.

Nel caso di chiave composta, bisogna definire una classe che, tratti della chiave e che definisca i metodi di confronto, per fare in modo che nel risultato della query appaiano dati doppi erroneamente.

La classe che possiamo vedere nella figura 3.2.3 e 3.2.4.

```java
public class InfoComunePK implements Serializable {

    /**
     *
     */
    private static final long serialVersionUID = -8739455792220522595L;

    private static final int MUL = 120;

    private static final int INIT = 132;

    private Long codcomune;

    private Long id_ccp;

    public Long getCodcomune() {
        return codcomune;
    }

    public void setCodcomune(Long codcomune) {
        this.codcomune = codcomune;
    }

    public Long getId_ccp() {
        return id_ccp;
    }
```

Figura 3.2.3:classe chiave InfoComunePK

```java
    public void setId_ccp(Long id_ccp) {
        this.id_ccp = id_ccp;
    }

    @Override
    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
        }
        if (obj == this) {
            return true;
        }
        if (!(obj instanceof InfoComunePK)) {
            return false;
        }

        InfoComunePK p = (InfoComunePK) obj;
        return new EqualsBuilder().append(codcomune, p.codcomune)
                .append(id_ccp, p.id_ccp).isEquals();

    }

    @Override
    public int hashCode() {
        return new HashCodeBuilder(INIT, MUL).append(codcomune).append(id_ccp)
                .toHashCode();

    }
```

Figura 3.2.4:metodi hashcode e equals dentro la classe

Permette di definire quali campi devono essere chiave, inoltre si avvale di due variabili (necessarie) MUL e INIT.

Infine nella parte finale della classe si definiscono i metodi EQUALS e HASHCODE, per spiegare alla classe come fare le distinzioni usando la chiave composta.

Se non si facessero questi passaggi non otterremmo mai un risultato veritiero dai dati del database, ogni altra classe dei metodi usati dal web service (che possegono una chiave composta), ha la sua relativa classe che descrive tale chiave.

Concludiamo la discussione di questo metodo facendo vedere anche come in questo caso, la mappatura dell'oggetto abbia generato delle modifiche.

Come si può notare il costruttore della classe Model, modifica dei campi (in questo caso intestazione), per dare una versione più conforme ai dati che deve ricevere il portale.

```
public InfoComune(Long codcomune, String comune, Long id_ccp,
        String sede_legale, String sede_operativa, String intestazione,
        String intestazioneente) {
    if (intestazioneente == null)
        intestazioneente = "Comune di ";
    this.codcomune = codcomune;
    this.comune = intestazioneente + " " + comune;
    this.id_ccp = id_ccp;
    this.sede_legale = sede_legale;
    this.sede_operativa = sede_operativa;
    this.intestazione = intestazione;
}
```

Figura 3.2.5: costruttore classe model

3.3 GETLOGOCOMUNE

Con questo metodo, otteniamo l'immagine del logo che si visualizzerà nel portale, logo che rappresenta l'ente/comune che deve essere risarcito.

Per prima cosa dobbiamo ottenere, tramite una query HQL il numero identificativo del conto corrente (ne discuteremo con il metodo GetLastAtto), questo numero, serve per indicarci il logo giusto, rappresenta per il database una chiave che distingue diversi tipo di atto e quindi di logo.

```
public DataHandler getLogoComune(long idratto, long codcomune,
        String tp_atto) throws DatabaseException {

    long idccp = 1;
    DataHandler logo = null;
    // mi serve prima l'id del conto per ottenere il logo giusto
    idccp = attoDao.getIdccp(codcomune, idratto, tp_atto);
    byte[] ba = logocomuneDao.GetLogoComune(codcomune, idccp);

    // restituisco l'immagine del logo ottenuto prima come array di byte
    logo = new DataHandler(new ByteArrayDataSource(ba,
            "application/octet-stream"));

    return logo;
}
```

Figura 3.3.1:metodo getLogoComune WS

```
package it.maggioli.informatica.wsgeris.daos.impl;

import it.maggioli.informatica.wsgeris.daos.LogoComuneDao;

@Repository
@Transactional
public class LogoComuneDaoImpl implements LogoComuneDao {

    @Autowired
    private SessionFactory sessionFactory;

    @Override
    // metodo per restituire un array di byte contenete il logo
    public byte[] GetLogoComune(Long codcomune, Long id_ccp) {
        Query q = sessionFactory
                .getCurrentSession()
                .createQuery(
                        "select logo from LogoComune where id_ccp = :id_ccp and codcomune = :codcomune");
        q.setParameter("id_ccp", id_ccp);
        q.setParameter("codcomune", codcomune);
        byte[] ris = (byte[]) q.uniqueResult();
        // sessionFactory.close();
        return ris;

    }
```

Figura 3.3.2:metodo getLogoComune DAOIMPL

L'oggetto che ci restituirà la query sarà un array di byte, che rappresenta l'immagine del nostro logo, se una colonna nel database è di tipo BLOB (quindi immagine), dobbiamo trattarla come array di byte(quindi in maniera generica).

Infine dovremmo trasformalo in un oggetto DataHandler per dare la possibilità al portale di portela trattare come immagine JPG o PNG.

Anche la classe database che definisce possiede una classe chiave, la chiave composta in questo caso è codcomune e id_ccp, questo perché alcuni comuni hanno più conto correnti e magari anche loghi diversi che rappresentano questi conti, quindi è necessaria una distinzione.

3.4 GETCONTRIBUENTE

Permette di ottenere le informazioni anagrafiche contenute, nel database del gestionale, tramite una query, utilizzando la classe Database che definisce la tabella nel Database vero e proprio.

```
/*
 * (non-Javadoc)
 *
 * @see
 * [redacted].wsgeris.WebServiceGeris#getContribuente(java.
 * lang.long, java.lang.long) metodo per ottenere le informazioni
 * anagrafiche
 */
@Override
public Anagrafe getContribuente(long cod_ctr, long codcomune)
        throws DatabaseException {

    // basta il codice comune e cod_ctr che sono le chiavi (nella query però
    // c'è anche il flattivo='A')
    it.maggioli.informatica.wsgeris.db.Anagrafe anag = anagDao.getAnagrafe(
            cod_ctr, codcomune);

    // mappo l'oggetto db nella classe modello
    Anagrafe ag = mapper.map(anag, Anagrafe.class);
    return ag;
}
```

Figura 3.4.1: Metodo getContribuente webservice

Il metodo in questo caso è molto più diretto, non vi è bisogno di grosse lavorazioni dei dati, basta solo fare la query attraverso la Classe Dao e restituire così, come vengono definiti nel gestionale, i dati del contribuente in oggetto.

Nel DAOIMPL, si definisce sempre una sessionfactory , che permette di eseguire la query. Il metodo getAnagrafe è derivato dall'interfaccia che ho sempre, costruito prima "ANAGRAFEDAO".

```
@Repository
@Transactional
public class AnagrafeDaoImpl implements AnagrafeDao {

    // serve x aprire la sessione x il db
    @Autowired
    private SessionFactory sessionFactory;

    @Override
    public Anagrafe getAnagrafe(Long cod_ctr, Long codcomune) {
        // hql
        Query q = sessionFactory
                .getCurrentSession()
                .createQuery(
                        "select c from Anagrafe c where cod_ctr = :cod_ctr and codcomune = :codcomune "
                        + "and attivo='A' ");
        q.setParameter("cod_ctr", cod_ctr);
        q.setParameter("codcomune", codcomune);
        // cast dell'unicio risultato che può dare la query
        return (Anagrafe) q.uniqueResult();
    }
```

Figura 3.4.2: Metodo getContribuente classe DAOIMPL

Il risultato sarà sempre unico e sarà veritiero in quando i dati inseriti in questa funzione derivano dal database.

La classe Database, definisce la tabella tabanagrafenew dentro lo schema riscossione, e la chiave che lo compone è data da codcomune e cod_ctr, che rappresentano l'identificativo dell'ente/comune e del contribuente all'interno del gestionale.

```
@Entity
// si da il nome della tabella/vista e lo schema
@Table(name = "tabanagrafenew", schema = "RISCOSSIONE")
// per gli id compositi si deve creare una classe e implementare con questa
// annotation
@IdClass(AnagrafePK.class)
public class Anagrafe {

    @Id
    // definisco il nome di colonna
    @Column(name = "CODCOMUNE")
    // definisco il nome della attributo che rappresenta la colonna nel db
    private Long codcomune;
```

Figura 3.4.3: classe database Anagrafe

3.5 GETLASTATTO

Questo metodo è di vitale importanza, in quanto come spiegato abbondantemente prima, anche se un contribuente possiede più codici derivati da vari atti consecutivi, questo metodo restituirà sempre l'ultimo atto che è stato ricevuto/elaborato.

```
@Override
public Atto getLastAtto(long idratto, long cod_ctr, long codcomune,
        String tp_atto) throws DatabaseException {

    String ultimo = null;
    // richiamo il metodo per ottenere la stringa che mi dira il tipo di
    // atto e l'id
    ultimo = ultimoattoDao
            .getLastAtto(idratto, cod_ctr, codcomune, tp_atto);

    String stpatto = "";
    // le prime due cifre e solo le prime due avranno il numerico che
    // identifica il tipo dell'atto
    Long tpatto = Long.parseLong(ultimo.substring(0, 2));

    // il tp atto è numerico ed è un range
    if (tpatto <= 20)
        stpatto = "I";
    else if (tpatto >= 21 && tpatto <= 29)
        stpatto = "S";
    else if (tpatto >= 30 && tpatto <= 39)
        stpatto = "F";
    else if (tpatto >= 40 && tpatto <= 49)
        stpatto = "G";
    else if (tpatto >= 50 && tpatto <= 59)
        stpatto = "P";

    // l'ultima parte della stringa contiene l'id dell'atto
    long idlastatto = Long.parseLong(ultimo.substring(3, ultimo.length()));

    // ottengo l'oggetto db Atto
    it.maggioli.informatica.wsgeris.db.Atto att = attoDao.getAtto(cod_ctr,

    Atto at = new Atto(att.getCodcomune(), att.getCod_ctr(),
            att.getImporto_residuo(), att.getImporto_pagato(),
            att.getDt_notifica(), att.getId_atto(), att.getStatus(),
            att.getTotale_atto(), att.getTp_atto(), att.getDestpatto(),
            att.getCcn(), att.getCci(), att.getCodiban(), att.getComune());
    // mapper.map(att, Atto.class);

    return at;
}
```

Figura 3.5.1:metodo getLastAtto webservice

Riceve in input l'id dell'atto (idratto), ottenuto grazie il metodo getAccesso, il cod_ctr che è sempre un tipo long, che rappresenta l'identificativo del contribuente nel gestionale e infine il tp_atto, una stringa che distingue i vari tipi di atto che sono:

- Ingiunzioni (il primo atto che viene spedito)
- Solleciti (semplice avvertimento spedito per atti sotto i 1000 euro)
- Preavvisi di fermo (atto che dichiara che si fermerà uno o più mezzi in caso di mancato pagamento)
- Precetti o intimazioni di pagamento (utilizzati per riaprire una pratica "dormiente")
- Pignoramenti

Questo metodo, innanzitutto deve verificare quale sia l'ultimo atto emesso, per farlo vi è una particolare vista contenuta nel database del gestionale, quindi basterà eseguire una query su essa per ottenere l'ultimo atto ricevuto dal contribuente.

Vengono usate due classe DAO AttoDAOimpl e UltimoAttoDAOimpl, nel primo DAOIMPL troveremo due metodi (vedi figura 3.5.3 e 3.5.4), getAtto e getIdccp, questultimo metodo che abbiamo visto essere usato nei metodi prima descritti.

```java
package it.maggioli.informatica.wsgeris.daos;

import it.maggioli.informatica.wsgeris.db.Atto;

public interface AttoDao {

    public Atto getAtto(long cod_ctr, long codcomune, long id_atto,
            String tp_atto);

    public Long getIdccp(long codcomune, long id_atto, String tp_atto);
}
```

Figura 3.5.2: interfaccia AttoDao

Partiamo descrivendo il metodo getIdccp, permette di avere l'id del conto corrente contenuto nel gestionale della riscossione, è necessario portelo conoscere per effettuare tutte le relazioni con i vari oggetti che andiamo a mostrare nel portale, senza questo non ci sarebbe integrità nei dati.

```java
@Override
// metodo per estrarre l'id del conto corrente dell'atto
public Long getIdccp(long codcomune, long id_atto, String tp_atto) {
    Query q = sessionFactory
            .getCurrentSession()
            .createQuery(
                    "select c.id_ccp from Atto c where codcomune = :codcomune and tp_atto = :tp_atto "
                    + "and id_atto= :id_atto");
    q.setParameter("codcomune", codcomune);
    q.setParameter("id_atto", id_atto);
    q.setParameter("tp_atto", tp_atto);

    return (Long) q.uniqueResult();
}
```

Figura 3.5.3:metodo getIdccp classe ATTODAOIMPL

La query restituirà sempre uno ed un solo risultato e verrà eseguita sempre utilizzando, la variabile di tipo sessionFactory per eseguire lo script della query e l'oggetto di tipo query, grazie inoltre alla funzionalità ".setParameter"si può dare valore alle variabili contenute nella stringa query precedente dal ":" .

```java
// metodo per ottenere le informazioni riguardo all'ultimo atto della storia
// della coattiva
public Atto getAtto(long cod_ctr, long codcomune, long id_atto,
        String tp_atto) {

    // hql
    Query q = sessionFactory
            .getCurrentSession()
            .createQuery(
                    "select c from Atto c where cod_ctr = :cod_ctr and codcomune = :codcomune "
                    + "and tp_atto = :tp_atto and id_atto= :id_atto");
    q.setParameter("cod_ctr", cod_ctr);
    q.setParameter("codcomune", codcomune);
    q.setParameter("id_atto", id_atto);
    q.setParameter("tp_atto", tp_atto);
    return (Atto) q.uniqueResult();
}
```

Figura 3.5.4:metodo getAtto classe ATTODAOIMPL

Ora trattiamo invece dell'altro metodo getAtto, utilizzato per avere informazioni dell'atto che si passa a questo metodo, contenuto nella classe AttoDAOimpl.

Si esegue una query sulla classe database Atto che andremo a vedere successivamente.

Il risultato sarà unico in quanto l'accoppiata codcomune, id_atto e tp_atto restituiscono sempre uno ed un solo atto.

```java
@Repository
@Transactional
public class UltimoAttoDaoImpl implements UltimoAttoDao {

    @Autowired
    private SessionFactory sessionFactory;

    @Override
    // ottiene stringa contenete nei primi 2 caratteri il tipo di atto e nei
    // rimanenti l'id dell'atto
    public String getLastAtto(long idratto, long cod_ctr, long codcomune,
            String tp_atto) {

        Query q = sessionFactory
                .getCurrentSession()
                .createQuery(
                        "select max(c.ultimo) from UltimoAtto c where codcomune = :codcomune and cod_ctr = :cc
                .setMaxResults(1);
        q.setParameter("cod_ctr", cod_ctr);
        q.setParameter("codcomune", codcomune);
        q.setParameter("idratto", idratto);
        q.setParameter("tp_atto", tp_atto);

        return (String) q.uniqueResult();
        // UltimoAtto lastatto =
        // return lastatto.getUltimo().toString();
    }
```

Figura 3.5.5: metodo getLastAtto classe UltimoAttoDAOIMPL

Trattiamo ora anche il metodo dell'altra classe DAOimpl (UltimoAttoDAOimpl)che viene richiamato getLastAtto.

Come spiegato prima ci possono essere più atti consecutivi e se qualcuno effettua il login con un vecchio codice dobbiamo dare tutte le informazioni dell'ultimo atto.

Questa query viene eseguita sulla classe UltimoAtto, classe che rappresenta una particolare vista dentro il gestionale della riscossione, che indica il tipo e il numero dell'ultimo atto spedito.

```java
@Entity
// si da il nome della tabella/vista e lo schema
@Table(name = "vwatti4web", schema = "RISCOSSIONE")
// per gli id compositi si deve creare una classe e implementare con questa
// annotation
@IdClass(AttoPK.class)
public class Atto {

    @Id
    // definisco il nome di colonna
    @Column(name = "CODCOMUNE")
    // definisco il nome della attributo che rappresenta la colonna nel db
    private Long codcomune;

    @Id
    @Column(name = "COD_CTR")
    private Long cod_ctr;

    @Column(name = "DA_PAGARE")
    private Double importo_residuo;

    @Column(name = "PAGATO")
    private Double importo_pagato;

    @Column(name = "DT_NOTIFICA")
    private Date dt_notifica;

    /*
     * @Column(name = "DT_SPEDIZIONE") private Date dt_spedizione;
     */
```

Figura 3.5.6: classe database Atto

Ora guardiamo le classi Database che vengono richiamate in questi tre metodi spiegati fino ad ora.

I metodi getAtto e getIdccp eseguono una query sulla classe Atto, la classe atto definisce la vista VWATTI4WEB, vista che descrive tutti gli atti presenti nel gestionale.

Nel caso vi steste chiedendo perché rappresentiamo una vista e non una tabella la risposta è che il gestionale usa in particolare quella vista per eseguire le ricerche e per mantenere l'integrità tra il web service e il gestionale, si è voluto usare la seguente vista perché in caso di modifiche sarebbe bastato modificare la vista VWATTI4Web per aggiornare entrambe le parti.

Anche questa classe usa una classe chiave e ovviamente le i campi chiave vengono marchiato con l'annotation @Id.

```java
// si da il nome della tabella/vista e lo schema
@Table(name = "VWLASTATTO4WEB", schema = "RISCOSSIONE")
// per gli id compositi si deve creare una classe e implementare con questa
// annotation
@IdClass(UltimoAttoPK.class)
public class UltimoAtto {

    @Id
    @Column(name = "COD_CTR")
    private Long cod_ctr;

    @Id
    // definisco il nome di colonna
    @Column(name = "CODCOMUNE")
    // definisco il nome della attributo che rappresenta la colonna nel db
    private Long codcomune;

    @Id
    @Column(name = "IDRING")
    private Long idring;

    @Id
    @Column(name = "TPATTO")
    private String tp_atto;

    @Id
    @Column(name = "IDRATTO")
    private Long idratto;

    @Column(name = "ULTIMO")
    private String ultimo;
```

Figura 3.5.7: classe UltimoAtto

GetLastAtto invece usa una particolare vista che nel gestionale ha il seguente nome, VWLASTATTO4WEB, il campo che ci interessa trattare a parte i soliti ormai conosciuti è il campo ULTIMO, una particolare stringa che contiene nei primi due caratteri un numero che identifica il tipo di atto, e nei restanti 12 caratteri il numero dell'atto nel gestionale.

Facciamo un esempio se ho 2000000000011, i primi due caratteri sono 20 e questo numero rappresenta un ingiunzione i restanti 00000000011, trasformati a numero diventa 11 quindi il risultato è l'ingiunzione numero 11, una volta conosciuto questo numero richiamando GetAtto(qua sopra descritto), avremo a disposizione tutte le informazioni dell'atto spedito.

NB anche questa classe ha una chiave composta formata da tutti i campi(vedi immagine), tranne da ultimo, senza di essa non avremmo il risultato.

Concludiamo la spiegazione del metodo del web service getLastAtto, dicendo che infine il risultato di questo metodo, dopo essere opportunamente mappato con la relativa classe Model, è solo uno in quanto al contribuente , per una determinata situazione coattiva corrisponde solo un atto attivo (gli altri atti hanno generato quest'ultimo).

3.6 GETSTORICO

getStorico, fornisce una lista di stringhe che descrivono precisamente da quale atto è partita la storia della coattiva e cosa è diventato successivamente.

```java
@Override
public List<Storico> getStorico(long idratto, long cod_ctr, long codcomune,
        String tp_atto) throws DatabaseException {

    // ottengo prima la lista di tutte le ingiunzioni iniziali che
    // compongono la storia della coattiva
    List<Long> idring = ultimoattoDao.getIdrIng(idratto, codcomune,
            cod_ctr, tp_atto);

    // una volta ottenute tutti gli id delle ingiunzioni che compongono la
    // sotira della coattiva
    // posso ottenere tutte le righe dello storico
    List<it.maggioli.informatica.wsgeris.db.Storico> storic = storicoDao
            .getStorico(idring, cod_ctr, codcomune);

    List<Storico> sto = new ArrayList<Storico>();

    for (it.maggioli.informatica.wsgeris.db.Storico s : storic) {
        // mappo l'oggetto db nella classe modello
        sto.add(mapper.map(s, Storico.class));
    }

    return sto;
}
```

Figura 3.6.1:metodo getStorico webservice

Il tipo di return in questo metodo sarà una lista di oggetti di tipo Storico ogni oggetto avranno due campi importantissimi per il portale, il campo stato e il campo prg, nel primo vi è una stringa che descrive il nodo dello storico nel secondo invece è il progressivo della storia, si parte da 1 fino a arrivare ad N.

Capita anche che in caso di contribuente molto moroso, che più storie coattive diventino una nuova e sola storia coattiva, in questo caso la classe storico, tratta anche questa situazione facendo vedere, sempre con precisione quale atto è stato seguito da un altro.

Per prima cosa in base al tipo di atto viene visto da quante e da quali ingiunzioni deriva esso, per fare questo usiamo un metodo contenuto dentro ultimoAttoDao, chiamato getidrIng.

Questo metodo esegue una query sulla classe database UltimoAttoDao e estrapola solo il campo idring, restituendo tutti quelli trovati.

```
@Override
// ottien tutti gli idr ing della storia della coattiva di un determinato
// atto
public List<Long> getIdrIng(long idratto, long codcomune, long cod_ctr,
        String tp_atto) {

    String squery = "select c.idring from UltimoAtto c where codcomune ="
            + codcomune + " and cod_ctr = " + cod_ctr + "  and tpatto = '"
            + tp_atto + "' and idratto= " + idratto;

    @SuppressWarnings("unchecked")
    List<Long> ris = sessionFactory.getCurrentSession().createQuery(squery)
            .list();
    return ris;
}
```

Figura 3.6.2:metodo getIdrIng classe StoricoDAOIMPL

Una volta conosciute tutte le origini si può ottenere tutto lo storico completo di questa situazione coattiva, utilizzando la classe StoricoDaoImpl (figura 3.6.3).

Verrà restituita una lista di tutte le posizioni dello storico, dove ognuna avrà la sua posizione PRG e la sua descrizione, il campo STATO.

```
import it.maggioli.informatica.wsgeris.daos.StoricoDao;

@Repository
@Transactional
public class StoricoDaoImpl implements StoricoDao {

    @Autowired
    private SessionFactory sessionFactory;

    @SuppressWarnings("unchecked")
    @Override
    // ottiene tutte le righe della tabistory per gli idring presente per la
    // suddetta storia coattiva
    public List<Storico> getStorico(List<Long> idring, Long cod_ctr,
            Long codcomune) {

        Query q = sessionFactory
                .getCurrentSession()
                .createQuery(
                        "from Storico where codcomune = :codcomune and cod_ctr = :cod_ctr  "
                        + "and idring in (:lidring) order by idring,prg");
        q.setParameter("cod_ctr", cod_ctr);
        q.setParameter("codcomune", codcomune);
        q.setParameterList("lidring", idring);

        List<Storico> ris = q.list();

        return ris;
    }
```

Figura 3.6.3: metodo getStorico classe StoricoDAOIMPL

NB in questa query è stata usato anche il comando in in cui è stato possibile passare una lista, infatti come si può notare dalla figura 3.6.3 vi è il comando setParameterList che riceve la lista di tutti gli id delle ingiunzioni.

La classe utilizza sempre una chiave composta (come si può vedere dalla figura 3.6.4 sotto) definita dalla classe chiave StoricoPK e descrive la vista VWSTORIA4WEB.

La relativa classe model è speculare a Quella database, quindi una volta ottenuti i risultati dello storico basta ri mapparla con la classe model, così il web service restituirà una lista ordinata (perché la query esegue una order by prg e idring) dello storico di quel contribuente.

```java
// si da il nome della tabella/vista e lo schema
@Table(name = "VWSTORIA4WEB", schema = "RISCOSSIONE")
// per gli id composti si deve creare una classe e implementare con questa
// annotation
@IdClass(StoricoPK.class)
public class Storico {

    @Id
    // definisco il nome di colonna
    @Column(name = "CODCOMUNE")
    // definisco il nome della attributo che rappresenta la colonna nel db
    private Long codcomune;

    @Id
    @Column(name = "COD_CTR")
    private Long cod_ctr;

    @Id
    @Column(name = "IDRING")
    private Long idring;

    @Id
    @Column(name = "PRG")
    private Long prg;

    @Column(name = "STATO")
    private String stato;

    @Column(name = "IDRATTO")
    private Long id_atto;

    @Column(name = "TPATTO")
    private String tp_atto;
```

Figura 3.6.4: classe database Storico

3.7 GETALLEGATI

Fornisce una lista degli allegati presenti per tutta la storia coattiva di una determinata posizione.

Il primo passaggio che esegue è quello di ottenere tramite getIdrIng tutte le ingiunzioni che sono presenti nella posizione con cui si è eseguito il login.

Una volta ottenute grazie a Getstorico verremmo a conoscenza di tutti gli atti che potrebbero esserci stati successivamente, con la lista fornita da GetStorico, potremmo eseguire un ciclo che richiama il metodo, getAllegati, in modo che esgua una query per ogni atto e formi una lista di oggetti Allegato (classe database), che andremo a "mappare", grazie al mapper e restituiremo a chi ha richiamato il metodo.

Se non ci fossero allegati disponibili, quindi zero elementi nella lista Allegati, verrà sollevata un eccezione usando sempre la classe DatabaseException, sollevando l'errore "Nessun allegato".

NB. Se la classe Database differisce dalla classe Model, bisogna fare un ciclo per ogni oggetto della classe Database, dove esegue una new (si richiama il costruttore) della classe del Model;

nel caso invece fossero speculari basta il metodo mapper.Map.

```java
public List<Allegato> getAllegati(long idratto, long cod_ctr,
        long codcomune, String tp_atto) throws DatabaseException {

    // ottengo prima la lista di tutte le ingiunzioni iniziali che
    // compongono la storia della coattiva
    List<Long> idring = ultimoattoDao.getIdrIng(idratto, codcomune,
            cod_ctr, tp_atto);
    // ottengo lo storico
    List<it.maggioli.informatica.wsgeris.db.Storico> storic = storicoDao
            .getStorico(idring, cod_ctr, codcomune);

    List<it.maggioli.informatica.wsgeris.db.Allegato> allegati = new ArrayList<it.maggioli.informatic

    // per ogni atto cerco i suoi possibili allegati
    for (it.maggioli.informatica.wsgeris.db.Storico s : storic) {
        allegati.addAll(allegatoDao.getAllegati(s.getId_atto(),
                s.getCodcomune(), s.getTp_atto()));
    }

    if (allegati.size() == 0)
        throw new DatabaseException("Error, allegati non trovati");

    List<Allegato> all = new ArrayList<Allegato>();

    for (it.maggioli.informatica.wsgeris.db.Allegato x : allegati) {
        // mappo l'oggetto db nella classe modello
        all.add(new Allegato(x.getId_atto(), x.getTp_atto(), x
                .getNomefile(), x.getNote(), x.getDescallegato(), x
                .getTipoallegato(), x.getPrg()));

    }
    return all;
```

Figura 3.7.1:metodo getAllegati webservice

Ora analizziamo il AllegatoDaoImpl e la relativa classe database Allegato.

```java
public class AllegatoDaoImpl implements AllegatoDao {

    // serve x aprire la sessione x il db
    @Autowired
    private SessionFactory sessionFactory;

    @SuppressWarnings("unchecked")
    @Override
    public List<Allegato> getAllegati(Long id_atto, Long codcomune,
            String tp_atto) {

        // hql
        Query q = sessionFactory
                .getCurrentSession()
                .createQuery(
                        "from Allegato where codcomune = :codcomune and id_atto = :id_atto "
                        + "and tp_atto = :tp_atto and tipoallegato in(2,3) order by prg");
        q.setParameter("id_atto", id_atto);
        q.setParameter("codcomune", codcomune);
        q.setParameter("tp_atto", tp_atto);

        // restituisco lista di risultati
        List<Allegato> ris = null;
        try {
            ris = q.list();
        } catch (Exception ex) {
            return null;
        }

        return ris;

    }
```

Figura 3.7.2:classe AllegatoDAOIMPL

```java
//definisce che è un entità
@Entity
// si da il nome della tabella/vista e lo schema
@Table(name = "VWTABALLEGATI", schema = "RISCOSSIONE")
// per gli id compositi si deve creare una classe e implementare con questa
// annotation
@IdClass(AllegatoPK.class)
public class Allegato {

    @Id
    // definisco il nome di colonna
    @Column(name = "CODCOMUNE")
    // definisco il nome della attributo che rappresenta la colonna nel db
    private Long codcomune;

    @Id
    @Column(name = "IDR")
    private Long id_atto;

    @Id
    @Column(name = "TPOBJ")
    private String tp_atto;

    @Id
    @Column(name = "NOMEFILE")
    private String nomefile;

    @Column(name = "PRG")
    private Long prg;

    @Column(name = "NOTE")
```

Figura 3.7.3:classe database Aleegato

NB. Anche la classe allegato ha la sua classe chiave AllegatoPK.

Vediamo anche il costruttore della classe Model, perché alcuni campi vengono modellati da essa e verrà già passati al portale nella maniera più corretta.

```java
public Allegato(Long id_atto, String tp_atto, String nomefile, String note,
        String descallegato, Long tipoallegato, Long prg) {
    this.id_atto = id_atto;

    this.tp_atto = tp_atto;

    this.nomefile = nomefile;

    if (tipoallegato == 2)
        this.note = "Copia dell'atto";
    if (tipoallegato == 3)
        this.note = "Copia della notifica avvenuta";

    this.descallegato = descallegato;

    this.tipoallegato = tipoallegato;

    this.prg = prg;

}
```

Figura 3.7.4:costruttore classe model Allegato

3.8 GETDOWNLOAD

Una volta ottenuta la lista degli allegati, il portale per ogni elemento di essa affiancherà a se un tasto per eseguire il download dell'allegato.

```java
/**
 * tipoallegato 2 -> pdf atto tipoallegato 3 -> cartolina
 */
@Override
public DataHandler getDownload(String fileDowload, long tipoallegato)
        throws DatabaseException {
    // oggetto generico per i file
    DataHandler dw = null;

    String spath = "";
    if (tipoallegato == 3)
        spath = getFilePathCart() + fileDowload;
    else
        spath = getFilePathPdf() + fileDowload;

    try {
        // creo un fileinput stream dando il percorso dell'allegato
        FileInputStream fileInputStream = new FileInputStream(new File(
                spath));
        // creo il mio datahandler dando il file input stream e un mime
        // type("application/octet-stream")
        dw = new DataHandler(new ByteArrayDataSource(fileInputStream,
                "application/octet-stream"));
    } catch (IOException e) {
        throw new DatabaseException("Errore nella lettura del file");
    }

    return dw;
```

Figura 3.8.1:metodo getDownload webservice

Il tasto in questione richiama questo metodo, ricevendo il nome dell'allegato e il tipo dell'allegato.

È richiesto anche il tipo perché in base ad esso, vi è un percorso fisico diverso dove raggiungere l'allegato.

Utilizzeremo le proprietà (definite in bean.xml e web.xml e successivamente dichiarate in WebServiceGerisImpl), che contengono il percorso dei vari allegati, una di queste proprietà più il nome dell'allegato ci danno il path da cui possiamo eseguire il download.

Download che caricherà il nostro risultato in un DataHandler che verrà dato in Output al portale.

3.9 GETVERBALI

Mostra l'elenco di tutti i verbali/accertamenti/fatture non pagati e che hanno generato atti nella coattiva.

```
public List<Verbale> getVerbali(long idratto, long cod_ctr, long codcomune,
        String tp_atto) throws DatabaseException {

    // ottengo prima la lista di tutte le ingiunzioni iniziali che
    // compongono la storia della coattiva
    List<Long> idring = ultimoattoDao.getIdrIng(idratto, codcomune,
            cod_ctr, tp_atto);
    // ottengo lo storico
    List<it.maggioli.informatica.wsgeris.db.Verbale> verb = verbaleDao
            .getVerbali(idring, cod_ctr, codcomune);

    List<Verbale> all = new ArrayList<Verbale>();
    for (it.maggioli.informatica.wsgeris.db.Verbale x : verb) {
        // mappo l'oggetto db nella classe modello
        all.add(new Verbale(x.getCod_ctr(), x.getCodcomune(), x.getIding(),
                x.getArea(), x.getVerbale(), x.getVerb_del(), x
                        .getVerb_noti(), x.getAnno(), x.getTarga(),
                getFilePathVerb()));
    }

    return all;
}
```

Figura 3.9.1:metodo getVerbali webservice

È molto simile ha getAllegati, infatti ha gli stessi passaggi, ovvero ottengo tutti gli Id delle ingiunzioni, grazie al metodo getIdrIng, dopodichè le metto come parametro di input nel metodo getVerbali della classe VerbaleDao.

Una volta ottenuta la lista di verbali/accertamenti/fatture eseguirò la mappatura con la relativa classe Model.

Vediamo ora la classe VerbaleDaoImpl, utilizza sempre le variabili utilizzate da tutte le altre classi DAOImpl e restituisce una lista di oggetti di classe database Verbale.

```
@SuppressWarnings("unchecked")
@Override
public List<Verbale> getVerbali(List<Long> iding, long cod_ctr,
        long codcomune) {

    Query q = sessionFactory
            .getCurrentSession()
            .createQuery(
                    "from Verbale where codcomune = :codcomune and cod_ctr = :cod_ctr  "
                    + "and iding in (:iding)");// order


    q.setParameter("cod_ctr", cod_ctr);
    q.setParameter("codcomune", codcomune);
    q.setParameterList("iding", iding);

    List<Verbale> ris = q.list();

    return ris;
}
```

Figura 3.9.2:metodo getVerbali classe Verbale DAOIMPL

La classe Verbale possiede una chiave composta e prende i dati dalla tabella tabdettaglio, come si può vedere dalla figura 3.9.3.

La chiave composta è definita nella classe VerbalePK.

```
// si da il nome della tabella/vista e lo schema
@Table(name = "tabdettaglio", schema = "RISCOSSIONE")
// per gli id composti si deve creare una classe e implementare con questa
// annotation
@IdClass(VerbalePK.class)
public class Verbale {

    @Id
    @Column(name = "COD_CTR")
    private Long cod_ctr;

    @Id
    // definisco il nome di colonna
    @Column(name = "CODCOMUNE")
    // definisco il nome della attributo che rappresenta la colonna nel db
    private Long codcomune;

    @Id
    @Column(name = "chiave")
    private Long chiave;

    @Column(name = "ID")
    private Long iding;

    @Column(name = "VERB_N")
    private String verbale;

    @Column(name = "VERB_DEL")
    private String verb_del;

    @Column(name = "VERB_NOTI")
    private String verb_noti;
```

Figura 3.9.3:classe databse Verbale

Una volta ottenuti i vari oggetti nel momento in cui si passa, dalla classe Database a quella Model vi è una manipolazione dei dati per quanto riguarda indicare il tipo di verbale, ovvero in base al campo area (che indica l'area dei tributi), il campo tipoVerbale deve assumere un valore distinto.

```
public Verbale(Long cod_ctr, Long codcomune, Long iding, Long area,
        String verbale, String verb_del, String verb_noti, Long anno,
        String targa, String path) {
    this.cod_ctr = cod_ctr;
    this.codcomune = codcomune;
    this.iding = iding;
    this.area = area;
    this.verbale = verbale;
    this.verb_del = verb_del;
    this.verb_noti = verb_noti;
    this.anno = anno;
    this.targa = targa;

    File f = new File(path + "\\" + codcomune + "\\"
            + verbale.replace("/", "") + ".xml");
    this.allegato = f.exists();

    switch (area.intValue()) {
    case 6:
        this.tipoverbale = "Verbale";
        break;

    case 11:
        this.tipoverbale = "Fattura";
        break;

    default:
        this.tipoverbale = "Accertamento";
    }
}
```

Figura 3.9.4:costruttore classe Model Verbale

3.10 GETDOWNLOADVERB

Simile a getDownoload, permette il download del verbali/accertamenti/fatture, ottenuto prima con il metodo getVerbali.

Utilizza la proprietà getFilePathVerb, per ottenere la radice del percorso fisico, dove risiedono questo tipo di allegati.

Inoltre è necessario, durante la costruzione del path, indicare l'estensione .xml, una volta ottenuto il percorso fisico eseguiremo il download dell'oggetto utilizzando un array di Byte per poi darlo in pasto, ad una variabile di tipo DataHandler.

L'array di Byte viene utilizzato per prendere file fisici, in quanto ogni file fisico è una lista di byte, il passaggio successivo con il Datahandler permette, di dare ha chi ha richiamato il metodo un oggetto, che potrà gestire come meglio crede.

```
@Override
public DataHandler getDownloadVerb(String verbale, long codcomune)
        throws DatabaseException {

    // oggetto generico per i file
    DataHandler dw = null;

    String spath = getFilePathVerb() + "\\" + codcomune + "\\"
            + verbale.replace("/", "") + ".xml";

    try {
        // creo un fileinput stream dando il percorso dell'allegato
        FileInputStream fileInputStream = new FileInputStream(new File(
                spath));
        // creo il mio datahandler dando il file input stream e un mime
        // type("application/octet-stream")
        dw = new DataHandler(new ByteArrayDataSource(fileInputStream,
                "application/octet-stream"));
    } catch (IOException e) {
        throw new DatabaseException("Errore nella lettura del file");
    }

    return dw;

}
```

Figura 3.10.1:metodo getDownloadVerb webservice

4 CASI D'USO

Andiamo ora a testare tutti i metodi presenti in questo web service, per farlo useremo la piattaforma di test SoapUI 5.2.0., della quale abbiamo già parlato prima.

Una volta compilato e avviato il web service abbiamo a disposizione (all'indirizzo di dove è posizionato il web service), il wsdl, prendendo il link del wsdl e inserendolo in un programma di test di SOAP UI, otterremo tutti i metodi disponibili, ora andremo a vederli nel dettaglio spiegandoli uno ad uno.

4.1 PANORAMICA GENERALE

i metodi sono i seguenti:

- getAccesso
- getinfoComune
- getLogoComune
- getContribuente
- getLastAtto
- getStorico
- getAllegati
- getDownload
- getVerbali
- getDownloadVerb

Tutti questi metodi danno la possibilità di far vedere al contribuente, tramite il portale, tutti i dati riepilogativi della sua situazione coattiva.

Vediamo nel dettaglio tutti i metodi sopra elencati, simulando per ognuno una richiesta dati SOAP.

4.2 GETACCESSO

Questo metodo permetterà il login da parte del contribuente al portale, bisognerà inserire un codice, presente in fondo all'atto che il contribuente ha ricevuto.

Con questo codice il web service saprà, tutte le informazioni della sua posizione coattiva, tipo le informazioni dell'atto, lo stato della posizione coattiva, a chi è destinato l'atto, chi spedisce e gestisce la coattiva della/e posizione/i non pagate in passato.

NB tutte le informazioni ottenute da questo metodo saranno essenziali per l'utilizzo dei metodi, successivamente descritti, senza queste informazioni, non si possono fare le richieste SOAP.

Il portale quindi si salverà le varie variabili e le riutilizzerà per ottenere le altre informazioni che il portale fa vedere.

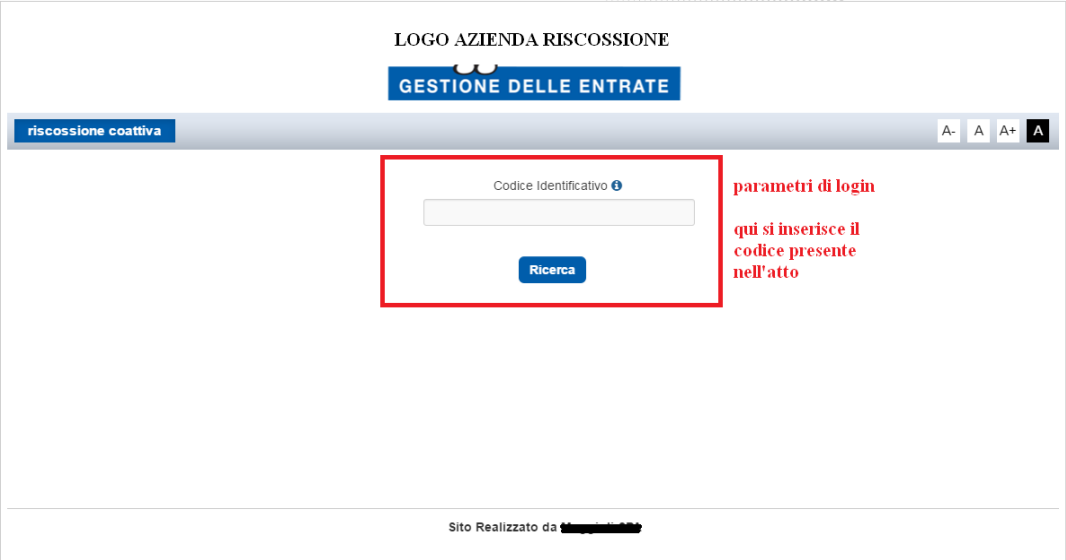


Figura 4.2.1: dove vengono inseriti i parametri di login nel portale

Il codice per effettuare il login è a 16 caratteri alfanumerici, in caso il codice non sia conforme a quella lunghezza il web service restituirà esito negativo;
un codice non conforme potrebbe essere perché non è lungo 16 caratteri, oppure è stato scritto in maniera sbagliata, inoltre il codice di login è case sensitive, quindi bisogna rispettare le lettere maiuscole e minuscole.

ecco come è stata simulata la richiesta in SoapUI

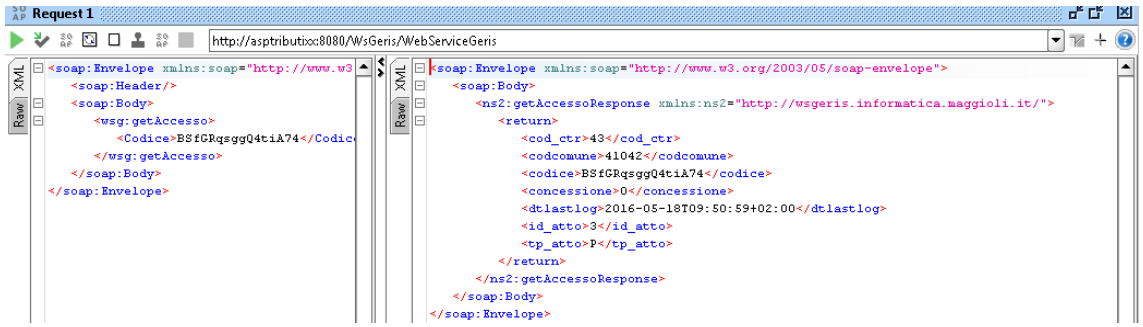


Figura 4.2.2:simulazione con soap UI

A destra dell'immagine abbiamo la risposta al metodo getAccesso, il risultato ci da esito positivo al codice inserito, in caso contrario avrebbe restituito un errore di login, inoltre ci restituisce informazioni utili che sono:

- **cod_ctr**: identificativo univoco per il contribuente nel comune in oggetto, dentro il gestionale della riscossione ogni contribuente, ha un codice identificativo, un proprio id.
- **codcomune**: è il codice ISTAT del comune anch'esso univoco è l'id del comune o dell'ente che viene utilizzato sempre nel gestionale.
- **codice:** viene restituito di nuovo il codice di login (informazione non più utile).

- **dtlastlog:** contiene la data dell'ultimo login effettuato nel portale usando quel codice, questa è un informazione che ci è utile sapere per delle statistiche interne nell'azienda, per sapere quanti accessi ci sono stati negli ultimi tempi.
- **id_atto** id univoco che identifica l'atto(da cui il contribuente ha ottenuto i codici x il portale), questo è l'identificativo unico con cui andremo a prendere tutte le informazioni relative a quell'atto.
- **tp_atto** definisce il tipo dell'atto, come spiegato prima la gestione della riscossione può essere effettuata in più fasi, nel caso il contribuente, continui a non pagare, ogni atto ha come attributo anche il tipo di atto, i vari tipi sono i seguenti:
    - I -> Ingiunzione (si parte sempre da questo atto nella coattiva e in caso di mancato pagamento con)i successivi che si vedono qui sotto)
    - S -> Sollecito
    - P -> Precetto
    - F -> Preavviso di Fermo
    - G -> Pignoramento

4.3 GETINFOCOMUNE

fornisce un insieme di informazioni utili su quel comune, richiede i seguenti dati **codcomune, id_atto, tp_atto**.

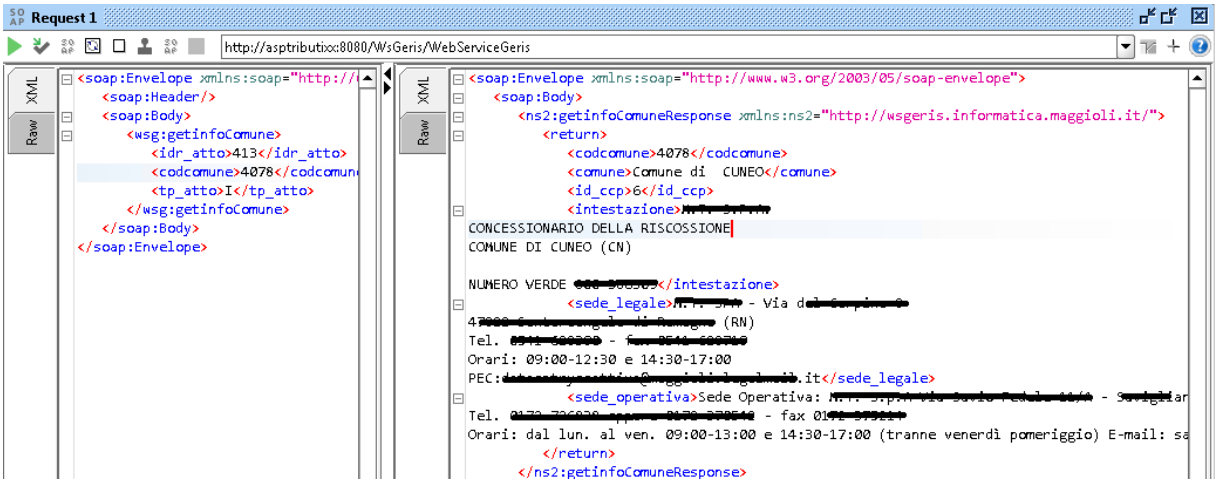


Figura 4.3.1:simulazione con soap UI

Analizziamo i campi nel dettaglio:

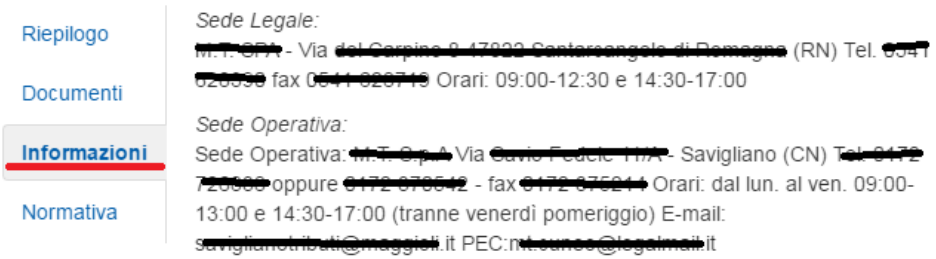
- **codcomune**: è il codice ISTAT del comune anch'esso univoco è l'id del comune o dell'ente che viene utilizzato sempre nel gestionale.
- **comune** nome del comune o dell'ente oppure della provincia oppure di un eventuale città metropolitana.
- **id_ccp** identificativo del conto corrente, dentro il gestionale, per ogni comune/ente, ci possono essere più conti correnti, dove il contribuente può pagare, il conto corrente dove bisogna pagare è specificato, nell'atto ma anche dal sito si potrà vedere per farlo abbiamo bisogno della chiave, dell'id, del conto corrente, così potremmo ripete le coordinate bancarie o postali dove regolarizzare la propria situazione morosa.

- **Intestazione** informazioni su chi ha spedito l'atto, ovvero chi si occupa della riscossione, questa è un informazione importantissima, qualora il contribuente volesse avere informazioni ulteriori o fare ricorso per quell'atto, dovrà fare riferimento a chi è presente in questa variabile di risposta SOAP.
- **sede legale** informazioni sulla sede legale, indirizzo e altre informazioni relative alla posizione della sede.

questa funzione è stata creata x essere utilizzata nel punto 3 dell'immagine:

Figura 4.3.2:prima pagina del portale dopo aver effettuato il login

E per riempire i dati, che potete vedere nell'immagine sottostante i quali, descrivono le informazioni sulla sede legale e sede operativa, di chi gestisce la riscossione di quell'atto.



4.4 GETLOGOCOMUNE

i comuni possono avere il proprio logo oppure può essere presente quello dell'agenzia di riscossione.

Inserendo i campi(già conosciuti all'accesso) , **codcomune, id_atto, tp_atto,** otterremo l'immagine, il marchio/logo, di chi ha effettuato la riscossione.

Questa immagine verrà utilizzata come logo del sito, quindi anche se il portale rimane unico il logo è variabile (figura 4.4.1).

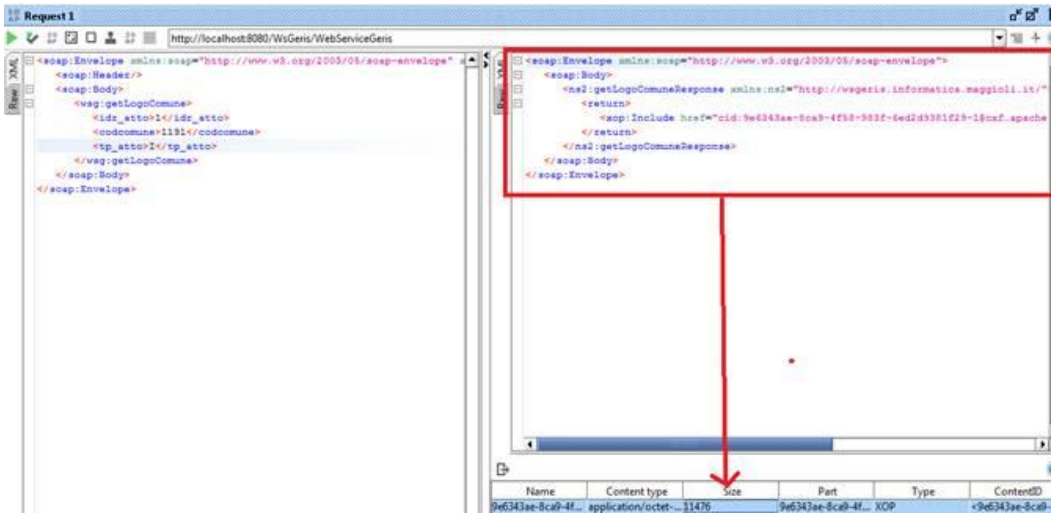


Figura 4.4.1:simulazione con soap UI

otterremo il download del logo corretto, il quale è sempre un immagine jpeg/png , che andremo a caricare nel punto (1) dell'immagine in figura 4.4.2, così ogni comune/ente potrà fare vedere il logo utilizzando tutti lo stesso portale.

Figura 4.4.2:prima pagina del portale dopo aver effettuato il login

Inserendo i campi(già conosciuti all'accesso) , **codcomune, Cod_Ctr,** ci permette di avere le informazioni anagrafiche del contribuente a cui è spedito l'atto, punto (4), tipo indirizzo, residenza , codice fiscale, data di nascita ecc.

Figura 4.5.1:prima pagina del portale dopo aver effettuato il login

Inserire solo queste due informazioni, il cod_ctr e il codcomune, si ottengo i dati anagrafici come possiamo vedere in figura 4.5.2.

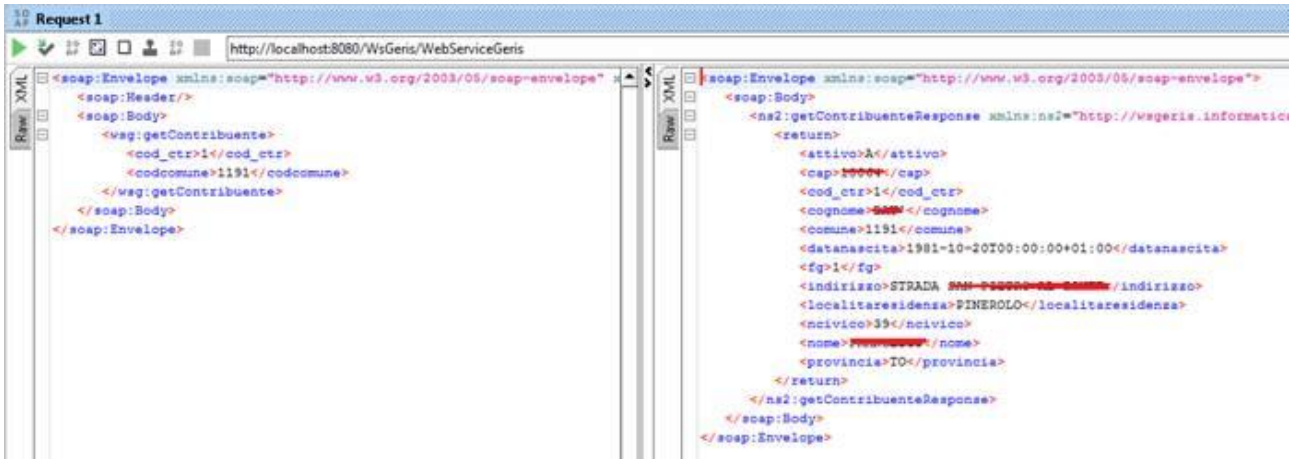


Figura 4.5.2:simulazione con soap UI con contribuente di tipo fisico

le possibili risposte sono due, in base a tipo di contribuente ovvero fisico o giuridico. Se il contribuente è fisico il campo **FG** è a 1 altrimenti a 2.

se il contribuente è giuridico, nella risposta del web service non ci sarà il campo nome ma solo il campo ditta con il nome del contribuente giuridico.

Figura 4.5.3:simulazione con soap UI con contribuente di tipo giuridico

Andiamo a vedere nel dettaglio i campi di risposta sia del Fisico che del giuridico.

FISICA:

- **Attivo:** se il contribuente è attivo nell'anagrafica
- **CAP:** cap di residenza
- **Cod_Ctr:** identificativo univoco per il contribuente nel comune in oggetto, dentro il gestionale della riscossione
- **Codcomune:** identificativo del comune nel gestionale della riscossione, viene utilizzato il codice ISTAT
- **Nome**
- **Cognome**
- **DataNascita**
- **F_G:** vale uno se è una persona fisica, 2 se giuridica (come spiegato prima)
- **Indirizzo:** indirizzo di residenza
- **Localitaresidenza:** città di residenza
- **Ncivico:** numero civico indirizzo di residenza
- **Provincia:** provincia di residenza

GIURIDICA

Ha gli stessi campi della fisica, cambia che non ha una data di nascita, in quanto persona giuridica e non possiede nome e cognome solo il nome della dittà/società/Partita iva.

- **Ditta:** il nome della ditta

si ottengo le informazioni sull'ultimo atto che riguarda la coattiva per determinati verbali, vedi (2) nell'immagine.

Questa funzione è necessaria perché nel caso un contribuente abbia ricevuto più atti relativi sempre ad uno o più accertamenti/verbali, nel caso provi ad utilizzare uno dei codici del portale di uno degli atti ricevuti, il programma indicherà sempre l'ultimo ricevuto.

Quindi il portale farà vedere (nel punto 2) il riepilogo all'ultimo atto ricevuto, questa funzione nasce per non fare confusione al contribuente, che non si troverebbe in grado di capire che cosa deve pagare, inoltre è molto utile nel caso si perda l'ultimo atto ricevuto ma si possiede un atto precedentemente ricevuto.

Figura 4.6.1:prima pagina del portale dopo aver effettuato il login

I campi richiesti per questa funzione sono :

- **idr_atto** : id dell'atto ottenuto grazie a getAccesso
- **cod_ctr** : codice del contribuente ottenuto grazie a getAccesso
- **codcomune** : codice del comune ottenuto grazie a getAccesso
- **tp_atto** : il carattere che indica il tipo di atto ottenuto grazie a getAccesso

In questa funzione ci sono varie informazione che potrebbero sono dentro il portale (vedi figura 4.6.2).

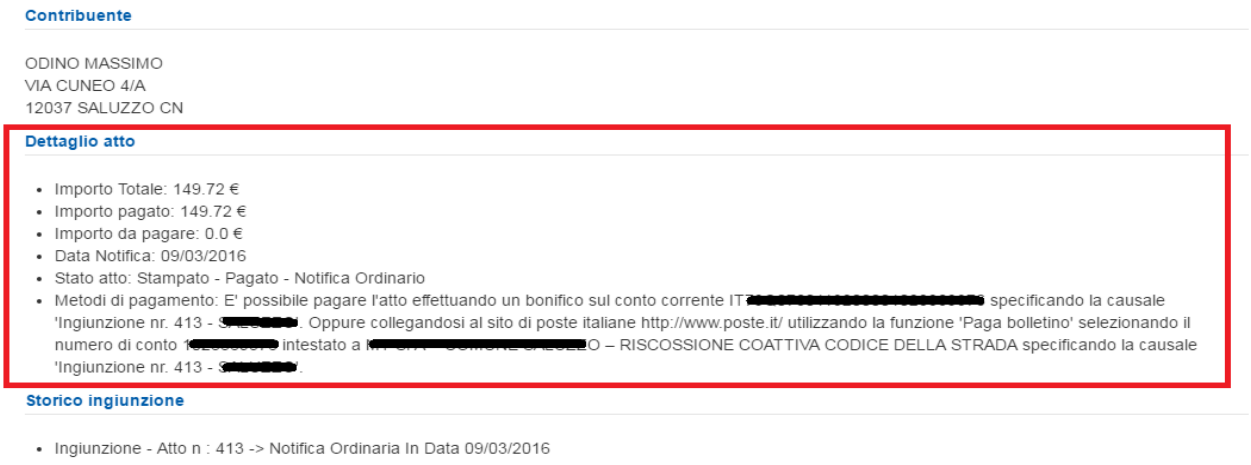


Figura 4.6.2:tutti i dettagli dell'ultimo atto ricevuto

Informazioni sempre utili per il contribuente, che indicano la situazione aggiornata all'ultimo atto; vengono così indicati quanto bisognava pagare, quanto si è pagato, l'eventuale residuo in più vengono visualizzati lo stato dell'atto e la data di notifica.

Vi è inoltre un ulteriore frase con la descrizione di come e dove pagare l'importo dovuto.

Andiamo a vedere nel dettaglio la richiesta SOAP.

Figura 4.6.3:simulazione con soap UI

E ora analizziamo i campi del risultato:

a parte quelli già trattati come codcomune, cod_ctr esaminiamo questi altri importanti dati :

- destpatto : indica in lettere il tipo dell'atto, es I è l'ingiunzione
- dt_notifica : data di quando è avvenuta la notifica dell'atto
- dt_spedizione : data della spedizione dell'atto (per ora questo campo non è visibile nel portale)
- importo_pagato : quanto è stato pagato dell'ultimo atto inviato
- importo_residuo : quanto rimane da pagare, in caso di pagamento parziale, viene fatto vedere quanto manca ancora da pagare, questo perché in caso di pagamento non conforme la storia della coattiva continua esigendo, il pagamento del residuo non ancora ricevuto.
- status : è una stringa riepilogativa sullo stato generale dell'ultimo atto inviato, indica sempre all'inizio la parola Stampato che indica che l'atto è stato fisicamente stampato ed inviato, le

successive parole indicano i vari stati che può aver preso quell'atto, per esempio può indicare la notifica oppure che è stato pagato oppure che ha subito un discarico.

- totale_atto : quanto si deve pagare, è la somma totale di tutte le parti dell'atto compresi eventuali oneri nel caso di una notifica CAD oppure una notifica CAN

4.7 GETSTORICO

ottiene una lista di tutti gli atti che formano la storia della coattiva.

Come spiegato prima la fase della coattiva parte con l'ingiunzione (tp_atto = 'I'), in caso di mancato pagamento, l'atto può diventare un altro atto successivo, con i riferimenti del precedente.

Quindi vi è una vera e propria storia da quando l'atto primario (ingiunzione) viene spedito fino all'effettivo pagamento o l'eventuale chiusura della pratica.

Nel gestionale della riscossione tutto questo, viene gestito vi è un vero e proprio cronologico che descrive il primo atto i successivi fino allo stato dell'ultimo atto inviato.

Facendo un breve esempio pratico abbiamo:

l'ingiunzione n° 3 diventa il sollecito n° 5 , perché l'ingiunzione non è stata pagata

il sollecito n° 5 diventa successivamente il preavviso di fermo n° 32, perché neanche questoè stato pagato

il preavviso n°32 ha stato Notifica Ordinaria, questo vuole dire che l'ultimo atto inviato è il preavviso di fermo 32 e che l'ultimo stato è una notifica ordinaria, cioè il contribuente ha ricevuto correttamente l'atto, se non pagherà anche questo atto la storia coattiva continuerà.

Per legge il contribuente deve sapere tutti i passaggi della coattiva che ha ricevuto, perché potrebbe fare delle verifiche, per conto suo o di un avvocato, per verificare la correttezza degli atti ricevuti; per esempio se volesse ricalcolare gli interessi o se volesse controllare i riferimenti normativi.

Per questo motivo nel portale diamo la possibilità di vedere tutti i passaggi, per dare chiarezza e trasparenza al contribuente, su tutti i passaggi.

Figura 4.7.1:simulazione con soap UI

il risultato della chiamata SOAP è una lista ordinata per progressivo (campo prg) in modo da avere l'ordine cronologico degli atti della coattiva, in ogni riga di return, in questo caso non si ottiene come risultato un unico campo, ma bensì una lista di tutti i passaggi della coattiva.

Il contribuente dovrà vedere principalmente solo il campo stato, ogni riga di stato è un atto della coattiva.

Analizziamo i campi nel dettaglio:

- **Cod_ctr :** trattato già in precedenza è l'id del contribuente nel gestionale
- **Codcomune :** codice ISTAT del comune l'id del comune nel gestionale
- **Idatto :** indica l'id (unico) che identifica l'atto nel gestionale della riscossione
- **Idring :** indica a quale ingiunzione, l'atto in esame è riferito, cioè se abbiamo un ingiunzione allora Idring sarà uguale ad Id, altrimenti ogni atto successivo avrà sempre il riferimento all'ingiunzione da cui è partita la storia della coattiva
- **Prg :** è il progressivo della storia, si parte da 1 e ogni atto successivo si fa +1
- **Stato :** indica lo stato dell'atto in esame che può essere, stato pagato, ricevuto(indica il tipo di notifica), discaricato oppure in caso di mancato pagamento indica il numero e il tipo di atto successivo che è diventato
- **Tp_atto :** indica il tipo dell'atto, come abbiamo già trattato prima se è un ingiunzione è uguale ad I se è un sollecito S ecc.

Figura 4.7.2:visualizzazione dello storico della storia coattiva vista nel portale

4.8 GETALLEGATI

Questo metodo permette di ottenere la lista di tutti gli allegati di tutti gli atti della coattiva riferiti ad una determinata posizione del contribuente, gli allegati possono essere la copia dell'atto ricevuto a caso oppure l'immagine della cartolina verde che attesta la notifica(e il tipo di notifica).

Funzione molte utile per dare la possibilità al contribuente di avere una copia dell'atto, in formato pdf, o dell'immagine della cartolina di avvenuta notifica. Può capitare che il contribuente perda la copia fisica dell'atto e ne richieda una copia, in questo modo può ottenere in qualsiasi momento tutte le copie che desidera, invece l'immagine della cartolina di notifica viene fornita nel caso il contribuente debba contestare una notifica, non conforme da quella scritta nella cartolina.

inserendo idr_atto, cod_ctr, codcomune, tp_atto ottenuti sempre dal metodo GetAccesso ,si ottiene la lista per ogni atto della coattiva.

Ecco un esempio di come si vede nel portale la risposta di questo metodo.

Figura 4.8.1:elenco degli allegati visibili nel portale

Ora andiamo ad analizzare questo metodo, facendo un test di prova con SOAP UI.

Figura 4.8.2:simulazione con soap UI

Analizziamo i campi nel dettaglio:

- **Id_atto :** l'id dell'atto nel gestionale della riscossione a cui è collegato l'allegato
- **Descallegato** : contiene la descrizione del tipo di atto a cui fa riferimento un determinato allegato
- **Nomefile** : serve per la funzione getDownload, contiene la posizione per ottenere il file fisico dell'allegato, questa funzione fornisce solo un elenco degli allegati, insieme alle coordinate di come ottenerli
- **Note** : si tratta di una frase di testo che descrive l'allegato
- **Tipoallegato** : indica il tipo di allegato che viene trattato, se il tipo è 2 si tratta di un atto in formato pdf, se invece il tipo è 3 si tratta dell'immagine dell'avvenuta notifica

Questa funzione, come quella che viene utilizzata per lo storico fornisce una lista di oggetti di tipo allegato.

4.9 GETDOWNLOAD

permette il download dell'allegato, vengono richiesti i campi nomefile e tipoallegato ottenuti con il metodo getAllegati, nell'immagine vista prima che faceva vedere nel portale la lista degli allegati, il metodo getAllegati fornisce quella lista che viene formattata nella tabella, di fianco ad ogni voce vi è un tasto download che permette di scaricare l'allegato.

Questo metodo fornisce il servizio di download, viene utilizzato il filepath per sapere fisicamente dove reperire l'allegato e tipo di allegato per fare distinzioni tra i pdf degli atti e le immagini delle cartoline.

Ecco l'immagine che mostra il test su SOAP UI per eseguire il download del file allegato.

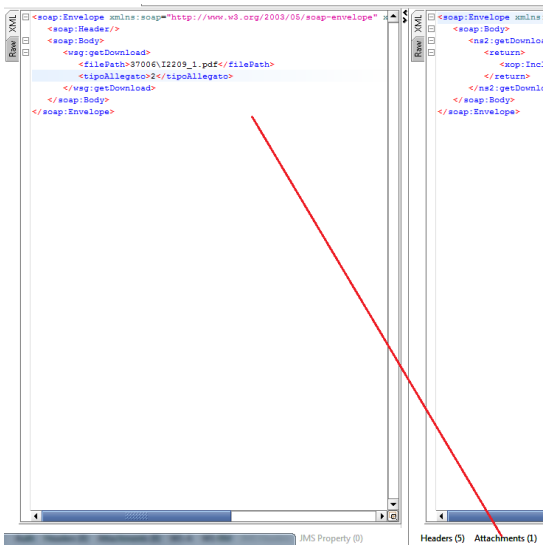


Figura 4.9.1:simulazione con soap UI con download dell'allegato

4.10 GETVERBALI

Questa funzione mostra tutti i verbali/fatture/accertamenti da cui è partita la coattiva.

In base al tipo di area tributi in cui ci troviamo, la parte della coattiva prende vita da questi 3 tipi di documenti:

- Verbali, nel caso un contribuente non abbia pagato una o più multe
- Accertamenti, nel caso un contribuente non abbia pagato una o più bollette, per esempio se non ha pagato la TASI o la tassa rifiuti
- Fatture, nel caso un contribuente non abbia pagato una o più bollette, in questo caso riferito ad un area tributi particolare della tassa rifiuti chiamata TIA

Nelle ingiunzioni quando parte la riscossione, ci sono i riferimenti a tutte queste posizioni non regolamentate, quindi è necessario dare al contribuente un elenco di cosa non ha pagato.

Sono sempre informazioni che devono essere fornite per legge per far capire al cittadino cosa non si ha regolarmente pagato, servono inoltre anche per far riferimento sempre a queste posizioni per un eventuale ricorso. Ovvero se io voglio contestare una multa che non ho pagato e che nel frattempo è diventata un'ingiunzione andrò dal giudice di pace indicano il verbale/accertamento/fattura.

Ecco quello che il contribuente vede nel portale.

Figura 4.10.1:elenco dei verbali/fatture/accertamenti vista nel portale

Nella chiamata SOAP inseriamo alcuni dei campi ottenuti con la funzione getAccesso, che sono idr_atto, tp_atto, codcomune, in questo modo SOAP UI restituisce l'elenco dei verbali/accertamenti/fatture che formano l'ingiunzione nella coattiva, insieme ad una serie di informazioni, tipo la data di emissione di questi documenti e la data di notifica di essi.

Figura 4.10.2:simulazione con soap UI

Analizziamo i campi nel dettaglio:

- **allegato** : se true vuol dire che è presente il file .xml del verbale/accertamento/fattura, file che descrive nel dettaglio che cosa andava pagato e i dettagli del documento, questo campo esiste perché, non tutti gli atti hanno questo tipo di allegato, ovviamente nel caso fosse presente bisogna dare la possibilità di visualizzarlo.
- **anno** : anno del verbale
- **area** : l'area è un campo presente nel gestionale della riscossione e rappresenta il tipo di area tributi che si tratta, ecco alcuni esempi di area dei tributi, se è uguale a 1 indica l'ICI , se uguale a 2 la tarsu , 5 all'acquedotto acquedotto, 6 il codice della strada, 10 all'area altre entrate patrimoniali, 11 alla TIA.
- **iding** : indica id (del gestionale) dell'ingiunzione di cui il seguente verbale/accertamento/fattura è contenuto.

- **targa** : questo campo è presente solo se siamo in AREA uguale a 6 ovvero codice della strada.
- **tipoverbale** : se CDS : verbale, se TIA : Fattura altrimenti Accertamento
- **verb_del** :data del del verbale/accertamento/fattura
- **verb_noti** : data notifica del verbale/accertamento/fattura
- **verbale** : contiene il numero del documento, questa informazione ci viene comunicata dall'ente/dal comune e rappresenta la posizione non pagata

se dopo aver fatto getVerbale otteniamo dei verbali con campo allegato uguale a true, possiamo eseguire il download, per farlo basta dare il codice comune e il campo Verbale ottenuto primo con la chiamata getVerbale.

Permette tutto ciò di ottenere un file, in formato xml, con tutte le informazioni del verbale, ovvero della o delle multe che non si sono pagate.

Il campo codcomune ci permette di sapere la posizione fisica di dove si trova l'xml dell'allegato e il campo verbale invece per trovare il file.

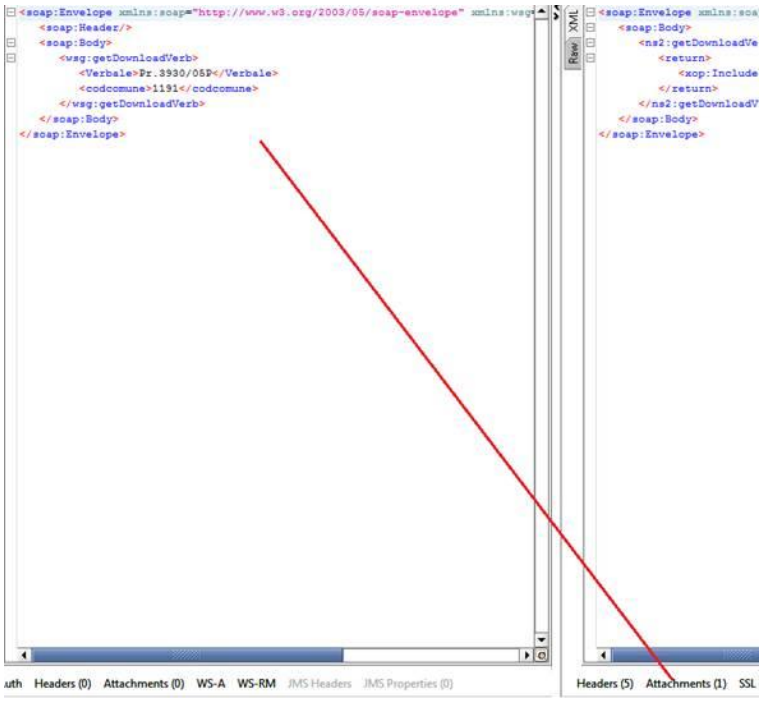
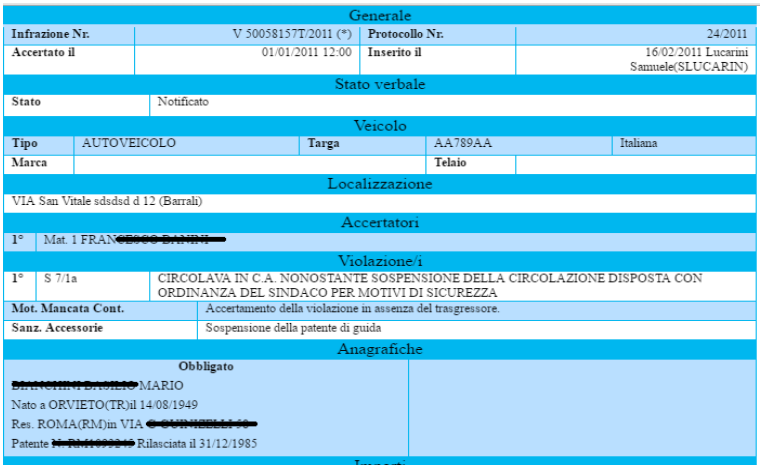


Figura 4.11.1:simulazione con soap UI con download dell'allegato

In questo caso non ci sarà un vero e proprio download da parte del portale, verrà visualizzato direttamente.
Questo per via del formato di questi verbali, che naturalmente non possono essere cambiati.



Generale			
Infrazione Nr.	V 50058157T/2011 (*)	**Protocollo Nr.**	24/2011
Accertato il	01/01/2011 12:00	**Inserito il**	16/02/2011 Lucarini Samuele(SLUCARIN)
Stato verbale			
Stato	Notificato		
Veicolo			
Tipo	AUTOVEICOLO	**Targa**	AA789AA Italiana
Marca		**Telaio**	
Localizzazione			
VIA San Vitale sdsdsd d 12 (Barrali)			
Accertatori			
1°	Mat. 1 FRAN[redacted]		
Violazione/i			
1°	S 7/1a	CIRCOLAVA IN C.A. NONOSTANTE SOSPENSIONE DELLA CIRCOLAZIONE DISPOSTA CON ORDINANZA DEL SINDACO PER MOTIVI DI SICUREZZA	
Mot. Mancata Cont.	Accertamento della violazione in assenza del trasgressore.		
Sanz. Accessorie	Sospensione della patente di guida		
Anagrafiche			
Obbligato			
[redacted] MARIO Nato a ORVIETO(TR)il 14/08/1949 Res. ROMA(RM)in VIA [redacted] Patente [redacted] Rilasciata il 31/12/1985			

Figura 4.11.2: esempio allegato verbale

5 CONCLUSIONI

L’obiettivo di questo progetto è stato quello di realizzare un web service che permettesse la possibilità ad un utente di poter vedere tutta la sua situazione morosa, in modo che comprendesse appieno tutti i passaggi che il gestionale della riscossione esegue, i punti realizzati sono stati:

- comprendere cosa il web service doveva fornire
- comprendere le tecnologie fornite per sviluppare il software
- fare in modo che il web service fornisse tutti i dati utili per il portale già esistente

È possibile affermare che tutte le specifiche richieste sono state soddisfatte e l’applicazione creata risponde perfettamente agli obiettivi prefissati.

5.1 SVILUPPI FUTURI

Il progetto creato rappresenta una buona base di partenza per possibili nuovi servizi legati alla riscossione, queste nuove utilità potranno permettere di :

- comunicare il contribuente direttamente dal portale
- dare la possibilità di pagare le proprie posizioni dal portale
- avere la possibilità di consultare ulteriori allegati agli atti
- chiedere un piano di rateizzazione direttamente dal portale

BIBLIOGRAFIA

[1] sito ufficiale APACHE CXF

http://cxf.apache.org/

[2] informazioni APACHE CXF

https://en.wikipedia.org/wiki/Apache_CXF

[3] guida all'uso di Spring framework

http://www.html.it/guide/guida-java-spring/

[4] guida hibernate html.it

http://www.html.it/articoli/introduzione-ad-hibernate/

[5] guida maven html.it

http://www.html.it/articoli/maven-organizzazione-dei-progetti-java-1/

[6] informazioni su Oracle

https://it.wikipedia.org/wiki/Oracle

[7] informazioni su linguaggio java

https://it.wikipedia.org/wiki/Piattaforma_Java

[8] informazioni xml

http://www.mrwebmaster.it/xml/introduzione-xml_9730.html

[9] informazioni eclipse

https://it.wikipedia.org/wiki/Eclipse_(informatica)

[10] sito ufficiale soap ui

https://www.soapui.org/

RINGRAZIAMENTI

Innanzitutto ringrazio tutti professori incontrati durante questi anni per i loro preziosi insegnamenti, in particolar modo il Professor Vittorio Maniezzo, mio relatore, per la professionalità e disponibilità dimostratami.

Desidero poi ringraziare i miei familiari, in particolare i miei genitori Casanova Andrea e Missiroli Silvana, che mi hanno sostenuto durante questo percorso.

Ringrazio tutti gli amici e i colleghi di lavoro, specialmente Davide Giorgi, che mi hanno sempre supportato, infine ringrazio la mia ragazza Valentina Colantonio per il sostegno e l'appoggio che mi ha sempre dato.