Alma Mater Studiorum · Università di Bologna

SECONDA FACOLTÀ DI INGEGNERIA
Corso di Laurea in Ingegneria Informatica

'il 730 online' di CAF ACLI e l'introduzione di una cultura DevOps

Tesi di Laurea in Ingegneria Informatica

Relatore:
Prof. Mirko Viroli

Presentata da:
Aduer Alessandrini

III Sessione
Anno Accademico 2017/2018

Sommario

In questo presente sempre più servizi che caratterizzano il nostro quotidiano (trasporti, utenze, commercio eccetera) stanno avendo una progressiva sostituzione della componente di interazione personale, in favore di una maggiore integrazione con *la rete*. Basti pensare a come grazie ad una connessione Internet, sia possibile monitorare in tempo reale il ritardo di un treno, o ordinare e ricevere a domicilio un piatto cucinato espressamente o vedere la sbarra del casello autostradale alzarsi perché il sistema ha riconosciuto la targa inserita poco prima tramite l'apposita app.

In contesti come questi, in cui un processo che avviene nella realtà, viene monitorato e coordinato dall'utente finale remoto, sul software grava una grande responsabilità.

Le software house per stare al passo con questa sfida, ed essere competitive sul mercato, devono produrre sistemi di qualità sempre maggiore, che sappiano dare soluzioni veramente utili agli utenti ed in tempi per loro accettabili. Questo può essere ottenuto adottando necessariamente metodiche di lavoro più sofisticate.

L'approccio *DevOps* in particolare mira ad ottimizzare i processi aziendali che intercorrono tra le varie fasi dello sviluppo di un software, sostituendo le attività manuali con procedure automatizzate capaci di aumentare la velocità, il livello di garanzia e la riproducibilità delle medesime.

Il lavoro presentato in questa tesi tratta di come questa metodologia sia stata introdotta per la prima volta nell'occasione della progettazione ed implementazione di un sistema di scambio di documenti fiscali tra contribuenti e il centro di assistenza fiscale.

Il sistema in oggetto è stato commissionato dall'ente di scala nazionale CAF ACLI all'azienda HI-NET di Rimini di cui sono dipendente al momento della presentazione di questa tesi e per la quale nell'occasione ho avuto l'onore di rivestire il ruolo di project leader.

L'obiettivo era quello di realizzare un servizio che si collocasse a metà via tra uno completamente telematico (come quello offerto dall'Agenzia Delle Entrate) ed uno tradizionale (commercialista di fiducia) prendendo il meglio che i due mondi avevano da offrire. Infatti, i benefici di un servizio online, quali la fruibilità 24/7 e la possibilità di operare senza recarsi presso uno sportello, uniti all'assistenza e garanzia del risultato che solo un esperto fiscale può dare, identificavano il servizio come una assoluta novità.

Sono state proprio queste componenti di innovazione, di rigorosità imposta dell'argomento e di dipendenza dal mondo legislativo italiano a rendere necessaria l'adozione di una metodica di lavoro più evoluta, che permettesse maggiore rapidità nei rilasci in modo da essere reattivi alle richieste di nuove funzionalità ed alle segnalazioni di bug.

Indice

Introduzione

Panoramica sul mondo delle dichiarazioni dei redditi in Italia

Ogni anno in Italia circa 17 milioni di contribuenti su un totale di 20 [1], si rivolgono ad un centro di assistenza fiscale per adempiere ai loro obblighi fiscali. La restante parte opera in autonomia, redigendo la propria dichiarazione dei redditi sul portale messo a disposizione dall'Agenzia delle Entrate. Negli ultimi anni questa forbice sta tendendo ad una progressiva stabilità, pertanto siamo davanti ad un dato strutturale non modificabile nella sostanza: **più dell'85% dei contribuenti preferisce affidarsi ad un CAF** per l'elaborazione della propria dichiarazione dei redditi.

Visti gli enormi numeri coinvolti, l'opportunità di business per le società che erogano questo tipo di servizi è grande ed ormai in Italia non esiste sindacato, confederazione o organizzazione del lavoro che non abbia una quota di *share* in questo mercato.

La mole complessiva dei clienti è immutabile (poiché funzionale al numero di soggetti aventi reddito) quindi l'unico mezzo a disposizione di queste società per sopravvivere alla competizione è quello di offrire un servizio migliore di quello della concorrenza ed intraprendere attività di marketing.

Le opportunità di miglioramento ci sono, perché tradizionalmente l'attività è per il cittadino particolarmente tediosa.

- Si tratta innanzitutto di un obbligo di legge, quindi anche nel più roseo degli scenari non viene svolto di buon grado da parte del contribuente.
- L'attività si svolge nel periodo estivo ma in riferimento ad una documentazione prodotta nell'anno precedente: il contribuente deve *sapere* quali sono i documenti utili, deve *collezionarli* per tutto il lasso di tempo necessario (circa 20 mesi) e deve saperli *conservare* in modo che si mantengano ben leggibili (es. scontrini).
- Deve prendere un appuntamento presso un ufficio fisico: vista l'alta domanda, difficilmente riuscirà ad ottenerlo in un momento favorevole.
- Deve fisicamente recarsi all'appuntamento con le ovvie implicazioni del caso (muoversi nel traffico, trovare parcheggio, fare la fila eccetera); questo problema è particolarmente sentito nelle grandi città.
- Per via dell'affluenza a questi sportelli l'operatore difficilmente riuscirà a dedicare molto tempo ad ogni pratica, quindi il servizio in certi casi potrebbe non risultare eccellente.
- Qualora emergesse qualche problema (es. documentazione carente o non idonea) tutto l'iter sarebbe da ripetere, con grande disappunto per l'utente finale.

La genesi de 'il 730 online'

Tra tutti gli utenti che si rivolgono agli sportelli fisici, la fetta dei pensionati è difficilmente conquistabile con nuove proposte, in quanto essi sono radicati nelle loro abitudini e sono insensibili alla tematica del risparmio del tempo. Per i giovani lavoratori ed i professionisti affermati il discorso è invece diverso. Questa fascia che comprende soggetti dai 25 fino ai 55 anni ha meno timore della novità ed ha interesse per una soluzione che permetta loro di ridurre lo stress ed ottimizzare questi aspetti documentali e logistici che ci sono dietro alla dichiarazione dei redditi.

CAF ACLI forte di questa convinzione, ha voluto aggiornare la sua offerta dotandosi di una soluzione che permettesse ai suoi 1,2 milioni di clienti [2] di ***farsi fare*** **la dichiarazione dei redditi dal loro ente di fiducia, ma senza uscire di casa**.

Analogamente alla proposta dell'Agenzia delle Entrate, l'utente non doveva più muoversi fisicamente con malloppi di carta, ma a differenza di quest'ultima non doveva nemmeno preoccuparsi di compilare le varie parti del modello: cosa che comporta l'elaborazione di complessi calcoli e l'assunzione della responsabilità su eventuali errori. La compilazione dei vari righi del modello sarebbe rimasta a carico dell'esperto fiscale (come nel tradizionale iter agli sportelli) ed il risultato analogamente garantito dal CAF. L'unica differenza avrebbe dovuto concretizzarsi nel modo in cui documenti ed informazioni sarebbero state scambiate tra contribuente e CAF.

Un approccio tecnologico rudimentale al problema (con uso di strumenti tradizionali come la email e gli allegati) non sarebbe risultato idoneo ad un impiego su larga scala e non avrebbe potuto garantire un sufficiente livello di privacy. L'operato del CAF è infatti soggetto alla sorveglianza da parte di diversi organi statali, quindi tutte le procedure che mette in campo devono essere eseguite con massimo rigore e sicurezza.

A grandi linee, sono state queste le forze che hanno spinto CAF ACLI a commissionare, per la stagione fiscale 2017, la realizzazione di un sistema che rispondesse a questa problematica e per il quale è stato scelto il nome di IL 730 ONLINE

Realizzazione

L'incarico dello sviluppo non è stato affidato ad una grande software house di scala nazionale perché nessuna di quelle interpellate inizialmente dal cliente gli avrebbe permesso un uso esclusivo del sistema sviluppato. Il progetto è stato infine commissionato all'azienda HI-NET di Rimini di cui sono dipendente al momento della presentazione di questa tesi.

Lo sviluppo ha richiesto circa 5 mesi per arrivare alla prima demo funzionante partendo dal primo incontro con il cliente. Poi nell'arco di un anno sono stati fatti numerosi aggiustamenti ed introdotte nuove features (es. la gestione delle pratiche IMU/TASI online).

Le varie attività sono state portate avanti da un piccolo team di 3 persone (me compreso) di cui mi sono ritrovato ad esserne *de facto* il team leader, per via degli incarichi e delle responsabilità di cui mi ero progressivamente fatto carico durante l'avanzamento dei lavori.

Altre attività minori (quali ad esempio il design grafico, il setup sistemistico degli ambienti ed il supporto help desk di 1° livello) sono stati affidati in parte ad altro personale interno dell'azienda ed in parte a risorse esterne o del cliente ma non saranno descritte in questa tesi in quanto di poco rilievo.

Relativamente allo sviluppo del sistema gli incarichi sono stati così suddivisi:

- Io ho coordinato lo sviluppo del progetto in ogni sua fase, partecipando a tutti gli incontri con il cliente, sin dalla fase iniziale di formazione in materia fiscale fino alla raccolta dei requisiti ed al brainstorming delle proposte.

  Ho realizzato l'analisi dei requisiti e fatto tutte le scelte tecnologico/architetturali, tenendo conto dei vari vincoli ed ottimizzando i compromessi in termini di adattabilità futura, robustezza, tempi di realizzo e fattibilità economica.

  Ho voluto l'adozione e sviluppato metodiche di DevOps, rendendomi ben presto conto di quanto il processo produttivo adottato inizialmente ci stava impedendo di lavorare in maniera efficiente. In particolare ho richiesto ed ottenuto dalla direzione l'installazione di un server interno aziendale da adoperare come sistema di version control e come esecutore di procedure automatizzate di rilascio. Questi argomenti sono trattati in dettaglio nel capitolo 3.

  Ho sviluppato integralmente la componente 'al pubblico' del sistema. Ho modellato le entità principali e sviluppato le logiche che ne governano l'interazione, e che sono condivise da tutte le componenti del sistema.

  Ho inoltre elaborato una metodica di testing automatizzato e scritto, con l'aiuto di qualche stagista di turno, centinaia di procedure di test.

- Un collega si è occupato integralmente dello sviluppo dell'interfaccia di backoffice, usata da amministratori, supervisori ed operatori del sistema. Si è anche occupato

di una parte del supporto di secondo livello (ossia della gestione delle segnalazioni di bug sollevate dal nostro cliente e non direttamente dagli utenti utilizzatori finali). Questi aspetti saranno solo marginalmente accennati in questa tesi.

- Un altro collega si è occupato principalmente delle relazioni con il pubblico. In particolare ha coordinato l'agenda di incontri ed appuntamenti con il cliente e con i suoi subordinati. Si è occupato della selezione dei fornitori esterni (quali ad esempio i provider di SMS telefonici, gateway bancari, consulenti privacy eccetera). Si è occupato della produzione della manualistica e della formazione del personale incaricato alla gestione del sistema. Si è inoltre occupato della redazione di report periodici da presentare al cliente a titolo di monitoraggio. Dal momento che la nostra azienda segue un percorso di qualità certificato, durante le varie fasi del progetto è necessario produrre della documentazione utile per rendicontare che i processi sono stati svolti attenendosi al disciplinare prestabilito. Anche questo aspetto è stato da lui seguito.

1 Background

In questo capitolo verrà fatta una rassegna delle nozioni tecniche e degli strumenti che è necessario conoscere per avere una corretta comprensione dei capitoli che seguiranno. Per via della grandezza del progetto, non è possibile fare una trattazione approfondita di tutte le nozioni, pertanto alcune saranno solamente accennate lasciando al lettore un eventuale approfondimento.

1.1 Nozioni di Ingegneria del Software

1.1.1 Patterns

Un *pattern* descrive un problema che si presenta frequentemente in un certo ambito ingegneristico e ne descrive il nocciolo della soluzione in modo che questa possa essere riusata.

Nell'ambito software, i pattern aiutano a costruire sistemi object-oriented di maggiore qualità [3].

1.1.1.1 Dependency Injection

Durante lo studio di un problema si cerca di affrontarlo scomponendolo in sotto-problemi più semplici. La soluzione che ne deriva è ottenuta assemblando soluzioni più elementari che avranno legami di *dipendenza* tra di loro. La corretta gestione di queste dipendenze è una problematica ingegneristica di rilievo, pertanto esistono dei patterns per gestirla.

La dependency injection permette un disaccoppiamento tra *entità dipendente* e *dipendenza* abbracciando i principi S.O.L.I.D [1]

Nella fase iniziale di vita di un sistema, a tutte le funzionalità viene associata una specifica concretizzazione (o comunque un'entità capace di erogarne una). Tutte le parti del sistema che hanno dipendenze su queste funzionalità non provvedono mai a soddisfarle autonomamente, ma si limitano a *dichiarare* che esiste questo vincolo. Nella fase

[1] *"one should depend upon abstractions, [not] concretions"* https://en.wikipedia.org/wiki/SOLID

successiva di esecuzione, un'entità denominata *injector* provvederà a risolverle *passando il controllo* all'entità dipendente solo quando questa sarà stata opportunamente dotata di tutto quello di cui necessita. La separazione degli ambiti viene così ottenuta con un accoppiamento delle parti parametrizzabile in fase di configurazione e non *hardcoded*.

1.1.1.2 Service Locator (o Dependency Fetching)

Quando non è possibile adottare la dependency injection (ad esempio per mancanza di features idonee nel linguaggio) esistono alternative.

Il pattern Service Locator (o anti-pattern a detta di alcuni autori [2]) permette di disaccoppiare le entità dipendenti dalle loro dipendenze ma a patto di spostare la dipendenza su un'entità mediatrice detta *container*.

1.1.1.3 Model View Controller

É un pattern architetturale molto diffuso che permette di disaccoppiare le logiche del sistema dall'interfaccia utente.

La *View* o *User Interface* propone all'utente una presentazione dello stato e dei dati del sistema ed elementi con cui può interagire per attivare certe determinate *Actions*.

Queste sono codificate nel *Controller*, allo scopo di tenere separati gli ambiti di presentazione dei dati da quelli di elaborazione delle richieste dell'utente.

Non sarebbe saggio codificare le logiche applicative nel controller perché essendo strettamente legato alla vista, risulterebbero transitivamente accoppiate a quest'ultima. Il luogo dove centralizzare la *domain logic*, il trattamento dei *dati* e le loro regole di *validazione* è il *Model*.

Esistono diverse accezioni di questo pattern per via dei diversi modi di interpretare i confini tra le parti. Per questo motivo spesso si parla di **Model-View-Whatever**

1.1.1.4 Object Relational Mapping & Active Record

É una tecnica che permette di colmare la sostanziale differenza che esiste tra un modello ad oggetti e quello di un database relazionale.

L'ORM maschera dietro semplici comandi le complesse query SQL che permettono di interagire con i servizi di persistenza tipici di un database (creazione, lettura, aggiornamento e cancellazione di record).

Un modo molto noto di mappare come oggetti i dati di un database è quello proposto dal pattern architetturale *Active Record*. Questo prevede di modellare le tabelle del database come classi ed i suoi record come istanze.

[2] *"couple code to the container [...] is known as the service locator antipattern"* http://php-di.org/doc/getting-started.html

Il fatto di lavorare su oggetti *in-memory* invece che direttamente sul database, permette di elaborare un numero minore di query, quindi di ridurre la latenza e migliorare la reattività del sistema.

1.1.2 Metodiche

1.1.2.1 Metodologia AGILE & eXtreme Programming

Sono strategie [3] [4] di gestione del processo di sviluppo e del coordinamento del team. L'idea al centro che le accomuna è che **il progresso può essere ottenuto stratificando micro-features**. Il fatto di avere cicli di sviluppo del software a raggio corto, anticipa gli eventuali problemi di integrazione permettendo di gestirli sul nascere e non a maturazione avvenuta.

1.1.2.2 DVCS

L'acronimo sta per Distributed Version Control System ed è una forma di sistema di *controllo versione*.

Con questo termine si intendono quelle attività che permettono la gestione di versioni multiple di un insieme di informazioni (quali ad esempio un progetto LaTeX, un disegno CAD o un programma per comupter).

Questo bisogno di gestire le revisioni esiste da quando esiste la scrittura ma nell'ambito dell'ingegneria informatica ha avuto un nuovo impeto, in quanto i sorgenti di un software sono sostanzialmente delle entità testuali.

Il funzionamento di questi sistemi permette di storicizzare i contributi dati dai singoli membri del team e permette loro di lavorare sugli stessi sorgenti contemporaneamente.

Ognuno di essi opera sua una propria copia locale della *versione* detta *working copy* e provvede ad allinearla periodicamente con il *repository* eseguendo dei comandi di contribuzione (es. *commit*, *push*) o comandi di prelievo (es. *clone*, *checkout*, *pull*).

Il repository può essere centralizzato e reso accessibile attraverso una rete, oppure distribuito.

Nel secondo caso ogni membro del team possiede una intera copia del *codebase* e della sua storia. Questo fatto permette di effettuare i comandi con maggiore velocità in quanto vengono eseguiti sulla copia locale e non attraverso la rete. L'aspetto distribuito permette inoltre di mantenere più repliche del sistema conferendo maggiore robustezza a fronte di incidenti.

[3] *Metodologia AGILE* https://it.wikipedia.org/wiki/Metodologia_agile
[4] *eXtreme Programming* https://en.wikipedia.org/wiki/Extreme_programming

1.1.2.3 Feature Branching

É una prassi [4] di impiego dei sistemi di version control che ha lo scopo di limitare o di ridurre l'entità dei problemi di fusione tra due filoni di sviluppo di un progetto (detti *branches*).

Consiste nel mantenere un branch *mainline*, in cui ogni versione rappresenta una release stabile del sistema. In questo modo è sempre possibile stratificare sopra veloci bugfix e rilasciare immediatamente, senza avere di intralcio features incomplete che ne precluderebbero la rilasciabilità. Queste features devono essere portate avanti su branch separati. I loro maintainer devono mantenerli allinearli con le evoluzioni avvenute sul filone principale durante il tempo del loro sviluppo. In questo modo ogni feature branch condivide la storia di quello mainline, e all'atto del *merge* non sorgeranno mai problemi.

1.2 Strumenti a supporto dello sviluppatore

1.2.1 Vagrant

É uno strumento[5] che sfruttando la tecnologia della virtualizzazione in ambito desktop[6] **permette di creare ambienti di sviluppo portabili e riproducibili in maniera svincolata dalla postazione di lavoro**.

Il funzionamento è basato sul processamento da parte di un tool automatico di un file testuale detto `Vagrantfile`. Questo file esprime le caratteristiche che deve avere la macchina ed i servizi che devono essere disponibili al suo interno.

Versionando il vagrantfile insieme ai sorgenti, ogni sviluppatore potrà averne accesso e grazie al tool Vagrant, ricrearsi senza sforzo un ambiente di lavoro ottimale per quello specifico progetto.

La fase nella quale i vari servizi vengono installati e configurati dentro la VM è detta *provision* e viene programmata con la **finalità di creare un ambiente il più rassomigliante possibile a quello in cui sarà rilasciato il sistema una volta ultimato**.

Questo permette sin dalle prime fasi dello sviluppo di anticipare il comportamento del sistema *in produzione* con una notevole fedeltà, migliorando la qualità del prodotto finale.

Lo strumento permette inoltre allo sviluppatore di lavorare dal suo sistema operativo di preferenza anche quando questo non corrisponde a quello target della soluzione che sta sviluppando (ad esempio potrebbe testare il comportamento di un sito Internet servito da un server UNIX, senza rinunciare a sofisticati software ad interfaccia grafica come Adobe Photoshop e Ms Excel).

La tecnologia in gioco permette di gestire delle cartelle condivise tra la macchina fisica *host* e quella virtualizzata. In questo modo i file possono essere acceduti anche dal sistema *guest* senza dover provvedere manualmente al trasferimento.

1.2.2 Package Managers

Spesso durante lo sviluppo di un sistema si incontrano problemi per cui esistono già soluzioni sviluppate da qualcun altro. Il web è pieno di librerie open source che risolvono le problematiche più disparate e che spesso hanno un pregevole livello qualitativo per via del contributo della community.

Tutto questo software *pronto all'uso* viene organizzato in *packages* e catalogato in registri pubblici detti *repository*.

Ogni repository è accompagnato da uno specifico **tool di gestione**, che permette allo sviluppatore di recuperare con facilità i pacchetti (e le loro dipendenze) semplicemente

[5] *Vagrant* https://www.vagrantup.com/

[6] *Virtual Machine* https://it.wikipedia.org/wiki/Macchina_virtuale

indicando in un file *manifest* il loro identificativo e la versione di cui necessita. Ne citiamo alcuni:

- `npm`[7] Viene distribuito insieme al runtime `node.js`[8] e dà accesso a quella che si blasona di essere (al momento della presentazione di questa tesi) la più vasta raccolta di librerie JavaScript esistente. La configurazione avviene attraverso il file `package.json`
- `bower`[9] Svolge le stesse funzionalità di npm, ma tradizionalmente viene usato per l'ambito frontend di progetti web. Si configura con il file `bower.json`.
- `composer`[10] É un gestore di dipendenze per progetti PHP che permette di installare i package del repository `packagist`[11]. Questo package manager si differenzia dagli altri perché oltre al file di definizione delle dipendenze `composer.json` (che indica quali solo le dipendenze e quali sono le versioni ammesse), affianca un ulteriore file `composer.lock` che indica esattamente quali versioni sono state installate l'ultima volta che il tool è stato lanciato da un qualunque membro del team. Versionando questo file tutti i membri condividono lo stesso setup indipendentemente da eventuali aggiornamenti rilasciati sul repository dei pacchetti.

1.2.3 IDE & Task Runners

Il linguaggio JavaScript, nato nel 1995, per moltissimi anni ha avuto un ruolo marginale nell'industria del software, in quanto utile solo per aggiungere logica alle pagine web. Quando nel 2009 è stato inventato node.js (un interprete di codice JS che opera al di fuori di un browser) è cambiato tutto.

Da quel momento c'è stata una **proliferazione di strumenti a linea di comando basati su JS** [6] e gradualmente il linguaggio ha conquistato piattaforme ed ambiti progettuali mai pensati in precedenza.

Gli IDE, ossia quegli strumenti che integrano editor del codice, compilatore, tool di build e debugger, semplicemente non hanno potuto più stare al passo con questa frenetica evoluzione e possiamo dire che ad oggi non ne esiste nessuno che offre queste funzionalità nativamente. Quello che fanno gli IDE più *web oriented* (come ad esempio `NetBeans`[12] o `Visual Studio Code`[13]) è fornire una interfaccia di coordinamento su vari tool del mondo node.js a linea di comando.

[7] *npm* https://www.npmjs.com/
[8] *node.js* https://nodejs.org/it/
[9] *Bower* https://bower.io/
[10] *Composer* https://getcomposer.org/
[11] *Packagist* https://packagist.org/
[12] *NetBeans* https://netbeans.org/
[13] *Visual Studio Code* https://code.visualstudio.com/

É molto difficile trovare un IDE che possa accomodare ogni possibile workflow di lavoro JS, pertanto **l'alternativa al tedioso lavoro manuale è l'automazione**.

Esistono numerosi *task runner*. Qui ne segnaliamo solo alcuni: `Gulp`[14], `Grunt`[15] e `Deployer`[16]. Quest'ultimo nasce con un intento diverso, ma nel bisogno può prestarsi anche come task runner.

1.3 Sviluppo di App

Con il termine *app* si intente una particolare forma di software che ha la caratteristica di essere *eseguibile* e di avere una forma di interazione con l'utente. Ad esempio un documento di testo può essere identificato come software, ma non come app, perchè non è eseguibile. Un driver di sistema è sia software che eseguibile, ma gli manca quella forma di *user interaction* quindi non è un app. A seconda del contesto, dello scopo e della tecnologia costruttiva, possiamo parlare di diversi tipi di app.

1.3.1 App Mobile

Una app mobile è un software specificatamente pensato per l'uso su dispositivi mobili quali smartphones e tablet.

Queste applicazioni sono progettate tenendo conto delle risorse limitate dei dispositivi che le eseguono (memoria, potenza computazionale, dimensione dello schermo, input gesturale) quindi hanno un paradigma di interazione specifico e funzioni limitate. Questo tipo di prodotti è mediamente pensato per un pubblico non evoluto tecnicamente, quindi hanno spesso un'interfaccia utente e delle funzionalità estremamente semplificate.

In circolazione esistono una gran moltitudine di dispositivi diversi sia dal punto di vista hardware che di supporto da parte del sistema operativo. Consegnare la giusta versione di un'app, per il giusto sistema posseduto dal cliente è un compito non banale. I grandi *player* di questo settore (Google ed Apple) hanno scelto di **rendere le app accessibili attraverso appositi *stores***.

Di fatto, per rendere un'app di dominio pubblico bisogna caricarla sugli stores, superando i protocolli da loro imposti ed ottenendo l'approvazione.

1.3.1.1 App Nativa

Sono applicazioni sviluppate specificatamente per una certa piattaforma mobile (`Android` o `iOS`).

[14] *Gulp* https://gulpjs.com/
[15] *Grunt* https://gruntjs.com/
[16] *Deployer* https://deployer.org/

Sviluppare applicazioni native comporta un grande sforzo e richiede un bagaglio di conoscenze molto ampio [7]. Ad esempio, per sviluppare applicazioni per la famiglia Android, lo sviluppatore deve avere familiarità con l'Android SDK ed il linguaggio Java. Per i dispositivi iPhone ed iPad invece occorre usare l'IDE Xcode ed il linguaggio Objective-C.

In aggiunta và considerato il fatto che ogni piattaforma ha la sua propria filosofia di sviluppo. Tutte queste sfide fanno sì che **sviluppare un importante app nativa per piattaforme multiple sia un'attività estremamente costosa**.

1.3.1.2 App Ibrida

Una app ibrida è un tipo di applicazione mobile che usa una finestra browser per mostrare la sua interfaccia grafica [8]. La tecnologia web è abbondantemente supportata su qualsiasi piattaforma mobile, quindi il fatto di effettuare lo sviluppo con comuni tecnologie web, permette di **ridurre lo sforzo ei costi per rendere la soluzione *cross-platform***.

L'interazione con i servizi fisici offerti dai dispositivi (geolocalizzazione, fotocamera, notifiche eccetera) è resa possibile grazie al progetto `Cordova`[17]. Questo mette a disposizione un'API invocabile da JavaScript che ha lo scopo di mascherare l'implementazione specifica di ogni piattaforma e permettere di *targettizzare* piattaforme multiple con un unico codebase.

1.3.2 Web App

Una Web App è un tipo di applicazione client-server la cui l'interfaccia grafica e la logica client side sono in esecuzione dentro ad un browser.

1.3.2.1 Round Trip Web Application

É il tipo di applicazione web tradizionale. Il termine *round trip* indica il percorso di elaborazione che viene seguito dal momento in cui l'utente effettua un'azione fino a quando ne vede l'esito a video. Questo comporta l'interrogazione di un nuovo URL da parte del browser, l'elaborazione sul server di una risposta e la consegna della medesima attraverso la rete. L'elaborazione lato server dona dinamicità ai contenuti, ma il prezzo da pagare è l'esperienza utente sacrificata dai continui *refresh* di pagina e dai tempi morti dovuti alla latenza della rete.

[17] *Apache Cordova - Phone Gap* https://cordova.apache.org/

1.3.2.2 Single Page Application

Una single-page application è un tipo di applicazione web che si differenzia da quelle tradizionali per il fatto che ad ogni interazione dell'utente il contenuto della pagina viene aggiornato senza effettuare un ri-caricamento completo, ma interessando solo le porzioni oggetto di cambiamenti. Le fasi di dialogo con il server avvengono in background senza interrompere l'esperienza utente e la mole di dati scambiati è minore perchè non viene effettuato un download completo dell'intera pagina. Il risultato è che la navigazione è più reattiva ed il carico sul server ridotto: questo aumenta la qualità di navigazione complessiva.

1.3.2.3 Progressive Web Application

Il termine `PWA`[18] è stato coniato da Google per identificare quelle web app che sfruttando certe funzionalità dei browser mobile di ultima generazione, possono essere percepite dall'utente come app native. I browser abilitati (come ad esempio Google Chrome) durante la navigazione di un sito/web-app fanno un'analisi sulla tecnica costruttiva e prelevano informazioni addizionali da un file `manifest` (quando questo è presente e referenziato dal codice html della pagina). Se la valutazione è positiva (nel senso che il browser reputa questa web-app degna di essere identificata come PWA) può proporre all'utente di *installarla* sul suo device. Questo farà comparire un'icona sulla *home screen* del dispositivo analogamente alle altre app native. I costruttori di tali applicazioni devono adottare le nuove tecnologie in maniera graduale, in modo da proporre sempre la migliore esperienza possibile all'utente: da qui deriva il termine *progressive*. Per misurare il livello di *compliance* della propria web app con i dettami del movimento progressive, Google mette a disposizione un tool[19].

[18] *PWAs* https://developers.google.com/web/progressive-web-apps/
[19] *Lighthouse* https://developers.google.com/web/tools/lighthouse/

1.4 Sviluppo Web

Con sviluppo web si intende quel lavoro di progettazione di siti Internet o applicazioni web. Questa attività spazia dal trattamento di dati, al fornire loro una rappresentazione e gestire una meccanica di interazione con l'utente, pertanto coinvolge una moltitudine di linguaggi, tecnologie e metodiche.

1.4.1 Linguaggi

Il grande successo del World Wide Web[20] si basa fondamentalmente sulla possibilità di collegare tra loro una grande quantità di ipertesti situati su macchine distanti attraverso l'utilizzo delle reti. Ciò avviene attraverso un protocollo di comunicazione detto `HyperText Transfer Protocol` che permette al server di rispondere alla richiesta del browser inviando un documento redatto in `HTML`, che è la notazione necessaria per scrivere i documenti che appaiono sul web. Sebbene semplice questo linguaggio ha grandi potenzialità comunicative perché permette di unire alla flessibilità dei collegamenti interattivi, la possibilità di formattare gradevolmente il testo e di introdurvi elementi multimediali. Negli anni il lavoro degli enti standardizzatori in materia è stato quello di promuovere la separazione tra *contenuto* e *veste grafica*[21], al fine di favorire la comprensione dei documenti anche da parte di robot automatici e non solo di umani. Il linguaggio che permette di codificare l'aspetto visuale è il `Cascading Style Sheet` mentre quello che permette di aggiungere interattività alla pagina reagendo ad azioni dell'utente e manipolando la stuttura del documento è il `Java Script`. Negli anni questi due linguaggi hanno subito notevoli aggiornamenti per stare al passo con le richieste tecnologiche del web moderno ma sostanzialmente non hanno mai cambiato il loro più profondo modo di funzionare. La stessa cosa è avvenuta nei browser che non hanno ampliato più di tanto le loro capacità di interpretazione di nuovi linguaggi. Per incontrare la domanda di maggiore espressività da parte degli sviluppatori sono nati i così detti linguaggi *transpillati*, ossia dei linguaggi che permettono allo sviluppatore di esprimere le sue volontà in un modo più evoluto per poi tradurre, o appunto 'transpillare', il codice in uno più basilare e interpretabile dal browser. Citiamo qui a titolo di esempio il linguaggio `Type Script`[22] che permette di scrivere codice *strong-typed* a differenza della sua controparte JavaScript in cui è convertibile. Per quanto riguarda invece lo styling segnaliamo il linguaggio `Less`[23] che grazie ad una sintassi gerarchica permette di esprimere le stesse regole stilistiche esprimibili con CSS ma scrivendo molto meno codice. Permette oltre di definire delle variabili ed eseguire operazioni aritmetiche in fase di conversione in modo da rendere il codice più leggibile e manutenibile.

[20] *History of WWW* https://webfoundation.org/about/vision/historyofthe-web/
[21] *Semantic Web* https://it.wikipedia.org/wiki/Web_semantico
[22] *Type Script* https://www.typescriptlang.org/
[23] *Less* http://lesscss.org/

Non sempre le risorse scambiate tra client e server sono statiche: in certi casi è necessario rispondere alla richiesta con un documento generato *al volo*. Per programmare sul lato server del sistema occorrono opportuni linguaggi. Il `PHP`[24] è uno di quelli più tradizionali, ma dalla release `5.x` ha visto l'introduzione di importanti features quali: supporto all'*Object Oriented Programming*, *namespaces* e *Lambda-Functions* tant'è che può essere identificato come un linguaggio di alto livello. Oltre al tradizionale formato HTML di scambio tra client e server, segnaliamo il formato `JSON`[25]. La sua semplicità ne ha decretato un grande successo, soprattutto in ambito di programmazione web asincrona (`AJAX`): di fatto il formato `XML` che dava la 'x' a questo acronimo (Asynchronous Javascript & XML) è stato in molti ambiti rimpiazzato.

1.4.2 Librerie & Frameworks

1.4.2.1 Yii Framework

É un framework[26] PHP object-oriented che permette di sviluppare rapidamente applicazioni web complesse favorendo l'adozione di sani principi progettuali.

Si compone di un corredo di classi base da usare come punto di partenza per sviluppare le proprie funzionalità e di utilities per aiutare lo sviluppatore nei compiti più disparati.

Dispone anche di alcuni tool a linea di comando per automatizzare le operazioni di *maintenance* più comuni sul progetto. Il tool `yiic` permette di eseguire comandi sviluppati dall'operatore: il suo uso principe è quello di applicare le modifiche strutturali al database relazionale eseguendo le query SQL di definizione delle tabelle, dette *migrations*[27].

Il *code generator* `Gii` distribuito insieme al framework, permette di scansionare il database dell'applicazione e di generare classi di *Model* che rappresentano le strutture dati adottando il pattern Active Record (1.1.1.4). Il tool Gii permette anche di generare rapidamente codice con funzionalità *CRUD* sui suddetti dati. Queste ottime funzionalità di manipolazione dati lo rendono un framework eccellente per sviluppare backoffice di amministrazione che non hanno troppe pretese estetiche e visuali.

L'altra caratteristica degna di nota è il modo in cui implementa il pattern MVC (1.1.1.3). Trattandosi di un framework di sviluppo lato server, non bisogna dimenticare che tutta l'esecuzione avviene nel contesto della generazione di una risposta (a partire da una certa richiesta HTTP). Il componente *router* del framework processa l'URL della request riconoscendo dei *tokens* nel seguente ordine: `<module><sub module><sub sub module><...><controller><action><resource_id>`

[24] *"What can PHP do?"* http://php.net/manual/en/introwhatcando.php
[25] *JSON* https://it.wikipedia.org/wiki/JavaScript_Object_Notation
[26] *Yii* https://www.yiiframework.com/
[27] *Database Migrations* https://www.yiiframework.com/doc/guide/1.1/en/database.migration

A questo punto il framework istanzia la classe *Controller* che identifica quella cascata ed esegue uno dei suoi metodi *Action* passando come argomento l'id della risorsa in oggetto. Questa organizzazione url-based delle logiche del sistema, permette di centralizzare a livello del modulo le funzionalità comuni a tutti i controller sotto la sua area di controllo. Ad esempio potremmo avere un modulo 'dashboard' che tratta le eccezioni di sistema presentando errori a video (output html) ed un modulo 'webservice' che le tratta ritornando al client opportuni response-code HTTP. Questo permette di sviluppare il codice delle action in maniera più uniforme e meno vincolato alla forma di output. Analogamente possono essere centralizzate politiche di caching, gestione delle autorizzazioni, redirect, policy di logging eccetera.

1.4.2.2 Firebase JWT

`Firebase` è una piattaforma promossa da Google per lo sviluppo di soluzioni web e mobile. L'autore e la community hanno sviluppato SDK per i principali linguaggi e sistemi operativi. Qui segnaliamo solo la libreria[28] PHP per la generazione di `JSON Web Token`[29]. I *JWTs* sono una tecnica per rappresentare *affermazioni* e veicolarle in maniera sicura tra client e server. Il loro impiego spesso è associato alla gestione delle sessioni utente. Invece che accompagnare la richiesta con un `cookie` (che deve essere mappato su una sessione lato server consumando risorse) il client conserva le sue informazioni di stato dentro al JWT ed usa quello per autorizzare la richiesta. Il meccanismo sta in piedi perché questi token sono creati usando sofisticate tecniche crittografiche ed ancora al momento della presentazione di questa tesi non si è trovato un modo per forgiarli facilmente da parte di malintenzionati.

1.4.2.3 AngularJS

É un framework per lo sviluppo di applicazioni web client-side promosso da Google. É stato progettato per permettere un ottimo disaccopiamento tra le logiche applicative e la manipolazione del `DOM` (ossia la struttura di oggetti che rappresentano un contenuto dentro la schermata di un browser) necessaria per operare aggiornamenti dinamici della pagina.

Attraverso la definizione di *directive* e *component* permette di estendere il linguaggio di markup introducendo nuovi tag personalizzati (es. `<my-calendar></my-calendar>`). A runtime questi tag vengono rimpiazzati con il rispettivo template html e ad esso vengono collegate le azioni codificate nel controller. Di fatto una web application di questo tipo è composta da una gerarchia di componenti. Il dialogo tra di essi avviene attraverso un protocollo di input-output ben definito, finalizzato a ridurre l'accoppiamento tra componenti diversi. Ognuno di questi può operare solo su dati e viste definite al suo

[28] *Firebase PHP JWT* https://github.com/firebase/php-jwt
[29] *JWT* https://jwt.io/

interno (il cosiddétto *inner scope*), mentre tutto ciò che viene scambiato con l'esterno (dati passati per copia, dati passati per reference, eventi emessi dal componente eccetera) devono essere dichiarati durante la fase di definizione del componente è sarà il framework a gestire il passaggio dei dati in maniera trasparente.

Il framework costringe lo sviluppatore ad impiegari solidi principi ingegneristici come la DI (1.1.1.1) ed il pattern MVC (1.1.1.3) rendendo la curva di apprendimento meno ripida anche per i meno esperti.

Per costruire una SPA (1.3.2.2) si sfrutta l'efficiente funzionalità di aggiornamento del contenuto offerto dal framework insieme ad una funzionalità di *routing*. L'applicazione funziona come una macchina a stati finiti: per ogni stato una specifica view và a sostituire il contenuto della pagina. La tradizionale navigazione tramite link ipertestuali viene sostituita da transizioni di stato. La libreria standard *de-facto* è `ui-router` di Chris Thielen[30] che è diventata popolare per via del fatto di risolvere le limitazioni del router nativo. Le versioni successive del framework hanno rotto la compatibilità con la prima nell'ottica di rendere il software indipendente dal contesto di esecuzione. L'app costruita in questo modo può funzionare anche al di fuori di un browser. La prassi del *JavaScript Isomorfico* [9] prevede la possibilità di eseguire l'app sia sul client (per offrire reattività all'utente) sia sul server (per dare una rappresentazione della pagina a quei crawler SEO, che tipicamente non intendono spendere loro risorse per computarla). Le versioni del framework successive alla prima invitano caldamente alla scrittura del codice col il linguaggio TypeScript [10] invece che JavaScript.

1.4.2.4 Bootstrap

`Bootstrap` è una libreria di stile CSS e di componenti JavaScript sviluppata dal team del social network Twitter. La sua caratteristica principale è il *column layout* che permette di suddividere la pagina orizzontalmente in sezioni fluide che possono cambiare nel loro numero o scalare la loro dimensione in base alla larghezza del monitor. In questo modo è possibile avere ad esempio un layout a colonna singola su schermi mobili (in cui i contenuti sviluppano nel senso verticale), mentre su schermi desktop organizzare diversamente lo spazio con una disposizione multi colonna. La feature del linguaggio CSS che permette questo tipo di cose è quella delle `Media Queries` ed il suo impiego per creare interfacce grafiche web che si adattano al device del navigatore prende il nome di *Responsive Design*.

1.4.3 API

Nel contesto dello sviluppo di applicazioni client side non è desiderabile permettere al client di accedere direttamente ai dati: questo creerebbe un accoppiamento forte che impedirebbe un corretto testing del sistema ed una facile evoluzione dello stesso.

[30] *ui-router* https://ui-router.github.io/ng1/

Interponendo il server come mediatore viene ad eliminarsi questo accoppiamento: la logica sul client responsabile di prelevare i dati non è a conoscenza di come questi sono effettivamente memorizzati ma ne riceve semplicamente una rappresentazione in un formato di scambio concordato. Lo strato di software che permette di rendere pubblico l'accesso a features del sistema viene tipicamente chiamato *Web API* ed esistono diversi stili e tecnologie per realizzarli.

Il protocollo `SOAP` permette tramite l'invio di messaggi, l'invocazione di procedure remote. Può operare attraverso diversi protocolli di rete sebbene quello HTTP sia il più diffuso. L'interfaccia del servizio viene descritta con un linguaggio formale basato su XML detto WSDL. Per ogni linguaggio esistono tool automatici capaci di confezionare un client apposito per un certo servizio partendo dal suo *manifesto* WSDL.

Lo stile `REST` prevede che sia una combinazione dell'URL e del metodo HTTP ad indicare l'operazione associata alla richiesta (inserimento, prelievo, aggiornamento, cancellazione).

Non esiste una specifica su come devono essere strutturati gli URL, ma l'idea di base è che questi siano *parlanti* [11] in modo che leggendoli sia subito riconoscibile qual'è la *risorsa* oggetto dell'operazione. Nel contesto di chiamate AJAX (1.4.1) il browser per motivi di sicurezza adotta la politica della *same-origin*[31]. Questa consiste nel permettere al codice client di effettuare richieste asincrone solo a server che hanno lo stesso dominio della pagina che si sta navigando. Quando il server ha un nome di dominio diverso, per far funzionare questa meccanica occorre implementare lato server una gestione di richieste dette di *preflight* che servono al browser per accertarsi che il server sia consenziente a farsi interrogare da codice operante su un dominio diverso.

[31] *Origin Policies* https://developer.mozilla.org/it/docs/Web/HTTP/CORS

1.5 Nozioni sul Testing

1.5.1 Unit Testing & End-2-End Testing

Nel contesto del testing automatizzato di sistemi software esistono diverse metodiche praticabili che si differenziano in base al tipo di **garanzie sul prodotto** che si vuole avere ed in base al livello di **sforzo tecnico che si vuole mettere in campo** per attuarle.

L'`Unit Testing` è la prassi secondo la quale i singoli elementi del sistema vengono presi uno alla volta e sottoposti a varie prove in isolamento. Questo metodo è tipicamente attuabile quando il team ha solide capacità di design, in quanto il fatto di far funzionare un pezzo di software in isolamento (e separato dalle sue dipendenze) richiede che i vari componenti siano stati progettati anche con questo scopo in mente. Inoltre, vista la grande quantità di *unità* presenti in un sistema 'grande', questo metodo richiede risorse umane in abbondanza per scrivere le numerose procedure di test ed avere un adeguato *test coverage*.

La prassi dell'`End to End` testing (spesso abbreviata con *e2e*) prevede il testing del sistema nel suo comportamento d'insieme. Nel contesto del web questo richiede un supporto tecnologico che permetta di **simulare un utente che compie delle azioni in un browser** (click, scroll, input di testo eccetera) e di percepire l'output delle sue azioni (comparsa di elementi nell'interfaccia, cambio dell'url, apertura di una finestra eccetera).

Con questo metodo si vanno a scrivere solo le procedure di testing che coprono gli scenari d'uso espressi in fase di raccolta dei requsiti: questo permette di garantire che *in un modo o nell'altro* il sistema faccia quanto promesso, ma senza avere un dettaglio tecnico di quanto è avvenuto effettivamente in background.

Il livello di capacità tecniche richieste per l'autore dei test e2e è estremamente alto perché simulare il comportamento di un utente (gestendo anche la variabile *tempo*) è un compito molto difficile. La scrittura di tests malfatti porterebbe ad avere esiti inattendibili (falsi positivi o falsi negativi). Citiamo qui alcuni termini:

- *Suite* Rappresenta un corredo di procedure di test.
- *Fixture* É una unità informativa da installare prima dell'esecuzione di un test al fine di creare il contesto necessario per il corretto esito di quest'ultima (ad esempio per testare il comportamento di un blog quando non ci sono articoli, occorre, che non ci siano articoli. La fixture in questo caso si occuperebbe di svuotare l'archivio).
- *Pattern Page Object* e *Pattern Step Object*[32] sono pattern che permettono di avere una visione di alto livello dell'attività compiuta in fase di test, disaccoppiando la

[32] *Reusing Test Code* https://codeception.com/docs/06-ReusingTestCode

suite dalle operazioni di basso livello effettivamente praticate. Ad esempio potremmo avere un Page Object che rappresenta la pagina dei login ed uno step object che rappresenta le azioni fatte da un utente guest e che ha un metodo 'doLogin'.

Per questo particolare progetto le suite end-2-end sono state scritte con l'ausilio del framework `Jasmine`[33] ed eseguite con il tool `Protractor`[34]. Questo tool è stato scritto specificatamente per Angular e sfruttando fatti noti del comportamento di questo framework, permette di eseguire le azioni previste dal test solo quando *la pagina è pronta* in seguito ad una modifica avvenuta. Il fatto che sia Protractor ad occuparsi automaticamente delle attese rende la scrittura dei test molti più semplice (rispetto ad altre soluzioni di *browser-automation* di più basso livello come WebDriver[35]).

1.5.2 Altri strumenti

Per eseguire i test a volte non basta impiegare una versione del software configurata per usare servizi *mock* ma a volte occorre effettuare una configurazione diversa nel sistema in cui è in esecuzione il codice.

Portando come esempio l'invio di email (che è una feature tipica di molti software e che spesso viene svolta dallo stesso server che esegue il codice) segnaliamo il mailer di sviluppo `Mail Catcher`[36]. Questo permette di spedire mail senza che queste escano effettivamente sulla rete pubblica ma allo stesso tempo permettendone di controllare l'avvenuto invio (che di fatto ha coinvolto tutto lo stack di rete visto che è basato sul protocollo SMTP).

Un'altra tematica tipica dei processi di testing è quella di disporre di dati fasulli ma plausibili. Il nostro sistema nelle fasi di iscrizione richiede dati all'utente e ne verifica la correttezza formale (cellulare, email, codice fiscale eccetera) Per permettere agli automi di compilare i form con successo abbiamo usato la libreria `Faker`[37]

[33] *Jasmine* https://jasmine.github.io/
[34] *Protractor* https://www.protractortest.org/#/
[35] *Selenium WebDriver* https://www.seleniumhq.org/projects/webdriver/
[36] *Mail Catcher* https://mailcatcher.me/
[37] *Faker* https://github.com/fzaninotto/Faker

1.6 Tematica CI/CD

1.6.1 Terminologia

Con i termini Continuous Integration, Delivery e Deployment [12] si intendono una serie di metodi di sviluppo software che prevedono l'esecuzione di una sequenza di procedure da parte di un agente automatico detto *runner* in risposta all'aggiunta di modifiche al codice in un repository centralizzato.

Le procedure coinvolte sono quelle tipiche del contesto delle *DevOps*[38] ossia quell'insieme di attività che occorrono per produrre un artefatto rilasciabile in modo che sia fruibile per l'utente, partendo dai codici sorgenti.

La sequenza di procedure prende il nome di *pipeline* e le fasi che la caratterizzano sono dette *stages*. Tipicamente gli stages sono: *build*, *testing* e *deployment*. Esiste anche una nozione di *environment* ossia di ambiente a cui è destinato il prodotto frutto dell'esecuzione della pipeline. Su un ambiente di pre-produzione (detto *staging*) potrebbe essere desiderabile avere a disposizione una versione del sistema che monta dei servizi mockati (ad esempio per usare un gateway bancario in modalità sandbox e testare procedure di pagamento senza spendere soldi veri). L'environment finale destinato all'utente è invece detto *production*.

1.6.2 Strumenti per deployment

La fase di deployment è quella che prevede la *posa in opera* dell'artefatto prodotto nella fase precedente di build in modo che sia fruibile dall'utente finale.

Nel contesto web questo comporta tipicamente l'upload di file su un server remoto e questa procedura può essere portata avanti con le più classiche soluzioni di trasferimento file (protocollo FTP, protocollo Rsync). In alcuni casi però può risultare necessario eseguire anche delle operazioni sull'ambiente remoto che non è possibile effettuare con un protocollo di trasferimento.

Un'esempio tipico è quello di aggiornare la struttura del database (es. aggiunta di tabelle) funzionale alla nuova versione del software che l'ambiente sta per ricevere. In questo caso bisogna coinvolgere dei protocolli più espressivi (es. SSH).

Questo fatto solleva un potenziale problema di sicurezza: l'addetto che si occupa dell'esecuzione delle *operazioni* deve disporre di credenziali di accesso che gli conferiscono grande potere, quindi è desiderabile far eseguire questi processi da tools automatici.

Qui menzioniamo nuovamente il tool `Deployer` (16) che ha una meccanica di funzionamento che incontra proprio questo scenario d'uso. Il tool accede via SSH all'host remoto, da lì clona i sorgenti dal repository e poi esegue una routine di preparazione specifica per il tipo di sistema che sta rilasciando (detta nel suo gergo *recipe*[39]).

[38] *DevOps e AWS* https://aws.amazon.com/it/devops/
[39] *Deployer Recipes* https://deployer.org/recipes.html

Il tool appena descritto è pensato per progetti PHP (il suo corredo di recipes copre i principali frameworks per questo linguaggio) ma non tutti i progetti sono fatti con questo linguaggio, quindi in generale occorre un grado di flessibilità più alto.

Non possiamo pretendere che il runner sia dotato di qualsiasi strumento in modo da poter eseguire qualsiasi procedura di deployment. Per aggirare questo problema viene usata la tecnologia della *containerizzazione* attraverso il tool `Docker`[40]. Questa permette di prelevare da un repository pubblico un ambiente pre-configurato (detto appunto *container*) che ha già a bordo i tool necessari per eseguire la procedura. Alla fine dell'esecuzione il runner rilascia il container e nessuna risorsa rimane impegnata su di esso.

1.6.3 Soluzioni integrate

Il processo di Continuous Integration coinvolge queste sfaccettature principali:

- Accesso al repository per ricevere una notifica quando è avvenuta una push su uno dei branch monitorati.
- Attivazione del runner per creare un ambiente containerizzato dove eseguire le fasi di build e poi di deployment.
- Gestire le chiavi di accesso necessarie per accedere sugli ambienti remoti.
- Fornire un feedback agli addetti sull'andamento del tutto.

Vien da se che con è possibile eseguire tutto questo in maniera artigianale ma serve una soluzione appositamente progettata per questi scopi.

`Travis CI`[41] è una soluzione completa ma è a pagamento per progetti non open source, quindi di fatto comporta un costo per le aziende che vogliono adottarlo.

`Jenkins`[42] è un prodotto della sfera Java e propone una integrazione perfetta con i tool caratteristici di questo mondo (Maven, Ant). Può essere impiegata anche per progetti diversi ma la configurazione è meno intuitiva.

`GitLab`[43] è una soluzione integrata che affianca ad una gestione Web based del repository e dei suoi utenti, potenti funzionalità di Continuous Integration. Può essere installata su un server standalone ed ha un approccio neutrale ai vari progetti gestiti. Offre inoltre la possibilità di gestire un wiki di progetto ed una gestione di *issues*. É la soluzione che abbiamo adottato.

[40] *Docker* https://www.docker.com/
[41] *Travis CI* https://travis-ci.org/
[42] *Jenkins* https://jenkins.io/
[43] *GitLab* https://about.gitlab.com/

2
il 730 online

2.1 Analisi dei requisiti

Il sistema commissionato non doveva essere un sistema per elaborare le dichiarazioni dei redditi, in quanto il nostro cliente CAF ACLI dispone già di una struttura informatica per questo scopo. **Lo scopo del progetto doveva essere invece quello di *dematerializzare* l'iter che avviene di persona presso i loro uffici** e che si articola attraverso questi passaggi:

- Il contribuente concorda un appuntamento presso uno dei tanti centro di assistenza fiscale sul territorio.
- All'appuntamento porta con sè tutta la documentazione utile per l'anno fiscale in oggetto: ossia quella prodotta nell'anno precedente a quello in cui sta effettuando la dichiarazione. Nel caso in cui nei due anni precedenti abbia presentato un dichiarativo (modello 730, modello unico eccetera) presso un CAF differente, deve portare con sè anche quel documento.
- L'operatore raccoglie il consenso al trattamento dei dati ed eventualmente all'invio di proposte commerciali.
- Se il cliente è nuovo provvede ad inserirne l'anagrafica nel sistema acquisendo anche una scansione del documento d'identità.
- Per integrare nel modello anche le informazioni disponibili per via telematica (es. tasse universitarie), l'operatore deve poter *operare in delega* sui portali del fisco ed ottenere l'accesso alla *precompilata* di quel cliente precedentemente elaborata dall'Agenzia Delle Entrate. I sistemi dell'agenzia consentono l'accesso ai dati del contribuente solo dopo il superamento di una procedura di verifica che prevede l'inserimento di alcuni valori particolari indicati nei dichiarativi degli anni precedenti. É questo è il motivo per cui il contribuente deve averli con sè nel caso in cui si sia rivolto ad altre strutture.

- A questo punto comincia la compilazione del modello: l'operatore vaglia la documentazione, digitalizzandola per conservarne una copia permanente nel suo sistema gestionale.
- Ogni anno in Italia le quote dell'8, 5 e 2 per mille dell'IRPEF[1] vengono devolute ad enti e partiti politici. Il contribuente può eventualmente esprimere una preferenza apponendo una firma sull'apposito riquadro di un modulo che gli viene fornito. Per certe scelte eventualmente può specificare il codice fiscale dell'ente beneficiario.
- A questo punto il ruolo del contribuente è terminato e può saldare il conto che può variare a seconda della complessità che ha richiesto per il calcolo.
- La dichiarazione viene trasmessa all'Agenzia delle Entrate per via telematica ma ciò non avviene immediatamente. Il CAF effettua un invio massivo poco prima della scadenza del termine di legge: in questo modo fino al quel momento il contribuente potrà modificare il suo dichiarativo integrando eventuali documenti dimenticati o segnalando errori che ha notato.

La documentazione coinvolta rientra assolutamente nella categoria dei *dati sensibili* in quanto dalle spese mediche si può risalire allo stato di salute del contribuente e dalle scelte di destinazione al suo credo politico-religioso. Quando si costruisce un sistema che tratta dati di questo tipo e su scala nazionale, occorre ottenere il benestare da parte del Garante della Privacy. Questo fatto ha introdotto nuovi requisiti ed aggiunto un notevole livello di complessità.

É infatti risaputo [13] che un sistema che vende servizi online, per avere successo, deve limitare il più possibile il numero di interazioni richieste all'utente. Per coniugare questo aspetto orientato al marketing, con le specifiche di rigorosità imposte, è servito un grande sforzo progettuale.

2.1.1 Requisiti Funzionali

Permettere di digitalizzare con facilità documenti cartacei Molti dei documenti utili per le finalità dichiarative non esistono in formato elettronico (es. rogito del notaio, scontrini della farmacia) mentre altri, sebbene generati partendo da una sorgente digitale, vengono rilasciati al contribuente solo in forma cartacea (es. buste paga, attestazione energetica dell'immobile). Per questo motivo il sistema doveva essere pronto a produrre una versione digitale del documento sfruttando qualsiasi strumento idoneo a disposizione dell'utente. Il fatto di poter *uploadare* direttamente files copriva gli scenari per i quali l'utente disponeva già del documento in formato elettronico o era dotato di uno scanner. Per tutti gli altri casi si doveva poter usare la **fotocamera del dispositivo mobile** come scanner. Questa

[1] *IRPEF* https://it.wikipedia.org/wiki/Imposta_sul_reddito_delle_persone_fisiche

funzionalità doveva poter essere richiamabile direttamente senza far uscire l'utente dall'applicativo.

Assegnamento operatori su base geografica Ogni contribuente al momento della sua registrazione viene affidato ad un operatore che lo seguirà per tutte le pratiche presenti e future su IL 730 ONLINE. Il criterio di assegnamento è basato sulla **provincia del contribuente**. Gli amministratori del sistema che inseriscono gli operatori, devono anche indicare quali sono le loro provincie di competenza; quando un nuovo cliente effettuerà la registrazione, il sistema sceglierà per lui l'operatore che copre la sua provincia che è attualmente *più scarico*. Se la specifica provincia non è assegnata a nessun operatore il sistema ne sceglierà allora uno qualsiasi in maniera random. Questa volontà di operare su una base geografica è stata rimarcata dal committente per far sentire ai suoi clienti un senso di vicinanza con il territorio.

Chat con operatore Per tutte le fasi in cui è richiesto dell'operato da parte dell'utente deve essere disponibile una chat con un operatore specializzato per permettergli di chiedere supporto. Il suo utilizzo sarà sempre gratuito e può essere usata a scopo consulenziale prima di iniziare l'upload dei documenti o anche in seguito per commentare quanto inviato e avere conferme.

Al contribuente non deve essere necessariamente dato di sapere chi è l'operatore con cui sta interloquendo: tutte le comunicazioni in arrivo devono essere firmate con CAF ACLI come mittente, mascherando l'effettivo soggetto. In questo modo, nella *chat room* tra contribuente e staff ACLI possono intervenire anche i supervisori dell'operatore o addetti appositi per le mansioni di *help desk* online.

Pagamento con carta di credito Il sistema non doveva offrire sistemi di pagamento tradizionali (come ad esempio il bonifico) perché avrebbero richiesto attività di gestione da parte degli uffici del CAF. Altresì era impensabile di implementare in casa una soluzione di pagamento elettronico: per questo tipo di attività bisogna ottenere la *compliance*[2] da parte dell'apposito ente, detto PCI.
L'unica strada percorribile era quella di scegliere un **fornitore esterno di servizi di pagamento elettronici**. Oltre ai tradizionali sistemi basati su carta recentemente ne stanno emergendo alcuni di nuovi che hanno la caratteristica di essere svincolati dai circuito bancario (es. Google ed Apple Pay, Stripe, PayPal). Tutti questi servizi sono erogati da grandi player internazionali quindi il fatto di adottarli avrebbe sollevato una questione sulla privacy. Bisogna poi aggiungere che non propongono commissioni particolarmente competitive quindi abbiamo deciso di scartarle a priori optando per un unico fornitore di serivizi di pagamento italiano.

[2] *PCI Compliance* https://it.pcisecuritystandards.org/minisite/env2/

Garantire la corrispondenza tra identità digitale e soggetto utilizzatore Tutti i principali sistemi online assicurano l'identità dell'utilizzatore con procedure di verifica della email o al più del numero di telefono. Un'approccio del genere non era adeguato per il nostro problema perché coinvolge dati sensibili. Abbiamo deciso di adottare come sistema di verifica l'inserimento di un pagamento simbolico di 1€ nella procedura d'iscrizione. Tale costo non inficia sul totale della prestazione perché sarà poi scontato al momento del saldo della pratica.

Un'accurato lavoro è stato fatto nella selezione del provider bancario in quanto il nostro requisito era quello di poter ridirezionare il contribuente al portale di pagamento con il campo *intestatario della carta* non editabile ma pre compilato con i dati inseriti da lui in fase di registrazione. Così il delicato incarico di effettuare **il matching tra identità fisica ed utenza su il 730 online veniva scaricato sul circuito bancario**. Quando un soggetto attiva una carta di credito si presuppone che l'istituito esegua idonei accertamenti sulla sua identità. Noi abbiamo deciso di **costringere l'utente ad effettuare il pagamento con una carta a lui intestata** per sfruttare proprio questo fatto.

Ricezione di notifiche Siccome il sistema prevede attività a lungo termine non è pensabile che gli interessati stiano ad osservare lo schermo in attesa di evoluzioni. Per quanto riguarda gli operatori serviva che fossero informati quando il contribuente aveva terminato di caricare i documenti ed aveva espresso le sue scelte di destinazione (cioè quando la sua pratica era pronta da elaborare). Per i contribuenti invece serviva informarli quando la pratica che avevano segnalato come *elaborabile* era stata *presa in carico* o era stata *elaborata*, quindi era pronta per essere *approvata* e *pagata*. Essendo il sistema targettizzato per l'intera popolazione italiana non potevamo scegliere una soluzione che funzionasse solo su dispositivi mobile. Abbiamo optato per l'**invio di email per notificare** gli avanzamenti di stato della pratica e per segnalare la presenza di eventuali comunicazioni nella chat rimaste non lette per un certo periodo.

Integrarsi con i flussi aziendali pregressi Soprattutto nelle prime fasi di lancio del sistema non si potevano dedicare troppe risorse umane per la sola piattaforma online, pertanto il sistema doveva essere progettato per inserirsi nelle dinamiche d'ufficio pregresse nella maniera meno invadente possibile. Questo requisito in particolare si è tradotto nel presentare una interfaccia di amministrazione che favorisse la migrazione dei dati verso il loro sistema di redazione dei dichiarativi.

Tracciare le attività degli utenti Per la promozione del servizio il committente ha investito del budget marketing su Facebook e Google. Anche se l'attività non era di nostro interesse, il sistema doveva essere integrato con le rispettive piattaforme in modo da fornire dati di tracciamento sui comportamenti degli utenti ai gestori delle campagne.

2.1.2 Requisiti Non Funzionali

Qualità e sicurezza Al momento del lancio il sistema era una assoluta novità nel settore, quindi un suo malfunzionamento avrebbe comportato un duro colpo in termini di immagine. Va poi ricordato che fà concorrenza ad un sistema governativo (costruito sicuramente con budget ben diversi) quindi ogni sua debolezza sarebbe ferocemente aggredita. Il fatto di trattare informazioni così delicate come redditi, malattie e credi personali avrebbe prodotto ripercussioni economiche gravissime in caso di fuga di dati o di hackeraggi. Tutti questi aspetti possono essere curati adottando metodi di lavoro che prevengano l'insorgenza di bug e **limitando il parco di versioni circolanti**. É infatti fondamentale fare in modo che eventuali bug, una volta scovati e corretti, non continuino ad esistere per via del fatto che alcuni utenti continuino a fruire di una versione obsoleta del sistema.

Cura del cliente Il sistema deve essere progettato per reagire prontamente ad eventuali segnalazioni di malfunzionamento da parte dei clienti finali. Questo si traduce nella capacità del team di **effettuare frequenti rilasci** e con poco sforzo. Se ad esempio un utilizzatore segnala un problema bloccante per lui, il team deve rilasciare una versione corretta del sistema in tempi accettabili per lui (ad es. in giornata) in modo da dare sempre un'**impressione di affidabilità e sicurezza**.

Gestione della tematica del 'tempo' Il sistema per sua natura ha a che fare con termini e scadenze di legge, quindi il suo comportamento in ogni stato è anche funzione della variabile tempo. Ovviamente eventuali malfunzionamenti che non ne permettessero il rispetto sarebbero inaccettabili.

Si ricorda che il dichiarativo viene elaborato l'anno successivo a quello dell'emissione dei documenti e la volontà era quella di affiancare il contribuente anche durante questa fase di 'cernita documentale'. Questo comporta che la pratica su IL 730 ONLINE sia già apribile svariati mesi prima di quando potrà essere elaborata. Questo fatto di *lavorare in anticipo* complica la situazione in quanto i decreti che regolamentano la materia non vengono emanati con così tanto anticipo, quindi il sistema deve **reagire in corso d'opera ai vari cambi di normativa**.

Performance Il fatto di poter usare la fotocamera del dispositivo mobile come scanner rendeva questa piattaforma meritevole di ricevere ottimizzazioni specifiche. La risoluzione media di uno scatto era di circa 5 Mega pixel, ma i telefoni di fascia più alta superavano già abbondantemente questa soglia con un trend in continuo aumento. Decidere di trasmettere lo scatto così come uscito dalla fotocamera, sarebbe stata una pessima scelta, principalmente per un fatto di **consumo della banda in upload**: nell'anno in cui abbiamo iniziato l'analisi ancora non erano molto diffusi piani dati mobile da svariati giga e non avremmo voluto trovarci davanti a clienti arrabbiati perché avevano prosciugato il loro traffico usando IL

730 ONLINE. I files ricevuti venivano per fini archivistici ulteriormente ridotti (formato bianco e nero, per un peso di circa 200kb per file) quindi sarebbe stato uno spreco trasmettere immagini ad alta risoluzione. Per questi motivi abbiamo scelto di **comprimere le immagini sul device** prima della trasmissione. Questa attività è molto onerosa computazionalmente e può essere deleteria sui device di fascia bassa, quindi serviva ottimizzare un compromesso tra requisiti di capacità di calcolo e risparmio del traffico dati.

Disaster Recovery In azienda questo argomento è sempre stato affrontato adottando soluzioni professionali di backup offerte dall'azienda Veeam[3]. In particolare queste permettono di elaborare automaticamente un'istantanea del sistema con cadenza giornaliera, settimanale, mensile e semestrale. Uno strumento del genere è utile per ripristinare il sistema ad uno stato funzionante ad esempio a fronte di hackeraggi o errori manuali degli amministratori ma non difende da errori del software. Se per qualche motivo la relazione tra gli allegati inviati dai contribuenti e le loro pratiche fosse andata perduta per via di bug non avremmo potuto risolvere con nessuna soluzione di backup al mondo. Per questo motivo abbiamo scelto di archiviare i file nel sistema con una organizzazione a livello di file system *parlante*. Le cartelle infatti rimandano all'identificativo del contribuente e le sottocartelle alla pratica, così anche nel peggiore dei casi è possibile risalire a *cosa-è-di-chi*.

[3] *Veeam* https://www.veeam.com

2.1.3 Requisiti Tecnologici

Poter funzionare su qualsiasi dispositivo Il fatto di sviluppare una versione del sistema web-based era imprescindibile. Il fatto invece di sviluppare applicazioni native per tutte le principali piattaforme avrebbe complicato enormemente il progetto sia in termini di costi (sarebbero serviti dei team specifici per ogni piattaforma ed un monte ore complessivo molto piu grande) che di affidabilità. Come già detto nelle trattazioni precedenti, per il rigore imposto dall'argomento era essenziale poter correggere bug tempestivamente e limitare il parco di versioni circolanti *buggate*. Questo non è facilmente praticabile per app native, perché i rilasci sugli app-stores richiedono tempo ed avremmo dovuto obbligare i clienti ad effettuare l'upgrade. La tecnologia web e le moderne features disponibili nei browser mobile avevano da offrire tutto il necessario per sviluppare questo sistema, pertanto abbiamo deciso di costruirlo come una **Single Page Application** (in modo da offrire un'esperienza di navigazione simile a quella di un'app) con **layout responsive** (per adattarsi a qualsiasi dimensione dello schermo) ed impiegando tecnologie costruttive che ne permettessero in futuro una **facile convertibilità in app ibrida**.

Potersi appoggiare all'infrastruttura hardware pre esistente Il committente disponeva già di un data center basato su tecnologia VMWare[4] in carico ad un'altra azienda. La richiesta era quella di **poter spostare il sistema** dopo il periodo di rodaggio iniziale presso tale data center. Questo requisito non ci ha creato problemi in quanto in azienda avevamo già esperienze pregresse di clienti che ad un certo punto avevano deciso di migrare il sistema presso i loro uffici o presso infrastrutture cloud, quindi siamo abituati a non fare supposizioni sull'effettiva architettura durante la fase di progetto.

Non serve che il sistema fosse SEO friendly La pratica della `SEO` è quella per cui il sistema viene appositamente disegnato per essere maggiormente apprezzabile dai motori di ricerca in modo da uscire tra i primi risultati quando un utente del web effettua una ricerca inerente.

Nel nostro caso tutte le attività di *posizionamento* e marketing erano svolte su un minisito apposito, pertanto la web-app **non era necessario che si indicizzasse**.

Questo fatto non è di poco conto, ed ha anche delle implicazioni tecnologiche considerevoli.

Una pagina web viene indicizzata per certe *keywords* quando il crawler del motore di ricerca al momento della sua visita trova contenuti sufficienti a farlo convincere che quella pagina è pertinente. Per le SPA (1.3.2.2) il rendering della pagina viene effettuato consumando risorse del visitatore, quindi sorge una problematica perchè

[4] *VMWare* https://www.vmware.com/it/solutions/virtualization.html

tipicamente i crawler non vogliono fare questo tipo di sforzo. Ciò che ne risulta è una totale non-indicizzazione perchè la percezione della pagina che ha avuto il crawler è quella di un *foglio bianco*.

Per risolvere queste problematiche si adottano sofisticate tecniche di rendering lato server ed hanno implicazioni profondissime sin dalle prime fasi del progetto e sulla scelta di tutto il parco tecnologico (1.4.2.3).

2.1.4 Requisiti Implementativi

Uso del legacy aziendale Vista la complessità del progetto e la mole di argomenti nuovi in ogni caso da affrontare, la direzione mi ha **vietato di introdurre nuove tecnologie** se non proprio dove fosse completamente impossibile impiegare quelle già note. Il resto del team non aveva dimestichezza con un uso avanzato del linguaggio JavaScript, quindi era impensabile pensare di coinvolgerli nello sviluppo della SPA o di adottare lo stesso linguaggio anche per l'elaborazione lato server (sebbene sarebbe stato un grosso vantaggio quello di condividere le entità principali). A grandi linee questo vincolo si è tradotto nelle seguenti specifiche:

- Uso del linguaggio **PHP** sul lato server e del framework **Yii 1.x** (1.4.2.1)
- Uso di **database relazionali**. Assolutamente vietato l'uso di database no-sql[5]. Su questa imposizione sono stato sostanzialmente d'accordo, in quanto i dati del problema sono effettivamente dati relazionati. In certi casi però (ad esempio per far fronte a cambi di normativa), la rigidità dello schema a tabelle è risultata leggermente di intralcio e ci ha portato ad adottare soluzioni intermedie quali ad esempio l'imbustare in un campo SQL testuale una struttura più flessibile JSON.
- Uso del web server di **Apache**[6]. Il reparto 'sistemi' dell'azienda non aveva esperienza con soluzioni alternative (es. NGINX[7]) e l'uso di Node.js [6] era praticamente da escludersi per i vincoli già elencati.

Supporto per testabilità In previsione del fatto che alcune fasi della procedura non sono testabili senza costi (es. pagamenti) occorreva prevedere sin dall'inizio di accoppiare i vari componenti del sistema sui *servizi* e non sulle loro implementazioni, in modo da poter passare da quelle effettive a quelle **mock** in maniera trasparente.

Inoltre per lo sviluppo di SPA basate sul router UI-Router (1.4.2.3) esiste un'utilissima estensione[8]; quando l'applicazione viene configurata per usarla, nella schermata dell'app, compare in sovra impressione un grafo degli stati che informa in tempo reale su qual'è lo *stato* in cui si trova l'applicazione e quali sono state le varie transizioni che l'hanno determinato.

Supporto per build multi-env Come intuibile dai precedenti punti, a partire dallo stesso *codebase* deve essere possibile **produrre versioni dell'applicazione che si differenziano** nell'uso o no di servizi mockati o strumenti di sviluppo.

[5] *NoSQL* https://it.wikipedia.org/wiki/NoSQL
[6] *httpd* https://httpd.apache.org/
[7] *nginx* https://www.nginx.com/
[8] *UI-Router Visualizer* https://github.com/ui-router/visualizer

Per altri progetti sviluppati in passato in azienda, questa tematica veniva affrontata modificando manualmente la configurazione del sistema un attimo prima del rilascio.

Questo approccio era prono ad errori e capitava di mettere in produzione una versione configurata per lo sviluppo, oppure di versionare i file di configurazione con parametrizzazioni specificatamente tagliate per supportare una qualche fase di testing.

Il fatto di gestire in maniera strutturata questa problematica ha richiesto di inserire un nuovo passaggio nella filiera di produzione del software. I linguaggi PHP e JavaScript sono direttamente processabili dai loro interpreti di riferimento, quindi una volta terminato lo sviluppo, ciò che si và a rilasciare sono direttamente i codici sorgenti senza alcuna trasformazione. Questa volta però, tra la fase di sviluppo e quella di esecuzione era necessario inserire una fase di *build* che permettesse di configurare il rilascio in maniera specifica per l'ambiente a cui era destinato. Nel capitolo 3 di questa tesi verrà discusso come questa ed altre fasi aggiuntive sono state strutturate in un processo organico ed efficiente.

2.1.5 Diagramma dei casi d'uso

2.1.5.1 Registrazione a il 730 online

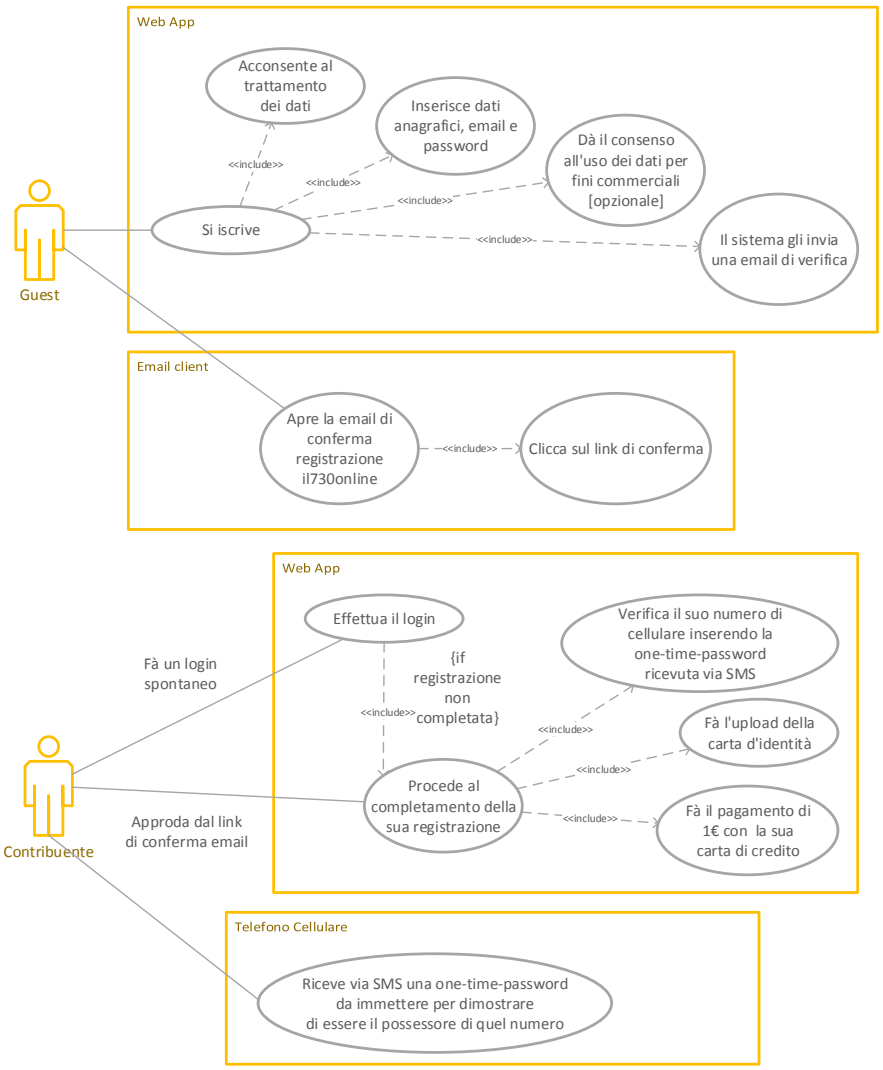

Figura 2.1: La registrazione avviene in due fasi: nella prima viene creata l'utenza per il contribuente (email + password), nella seconda si verifica che l'identità digitale corrisponda a quella del soggetto fisico che vi è dietro.

2.1.5.2 Apertura pratica 730

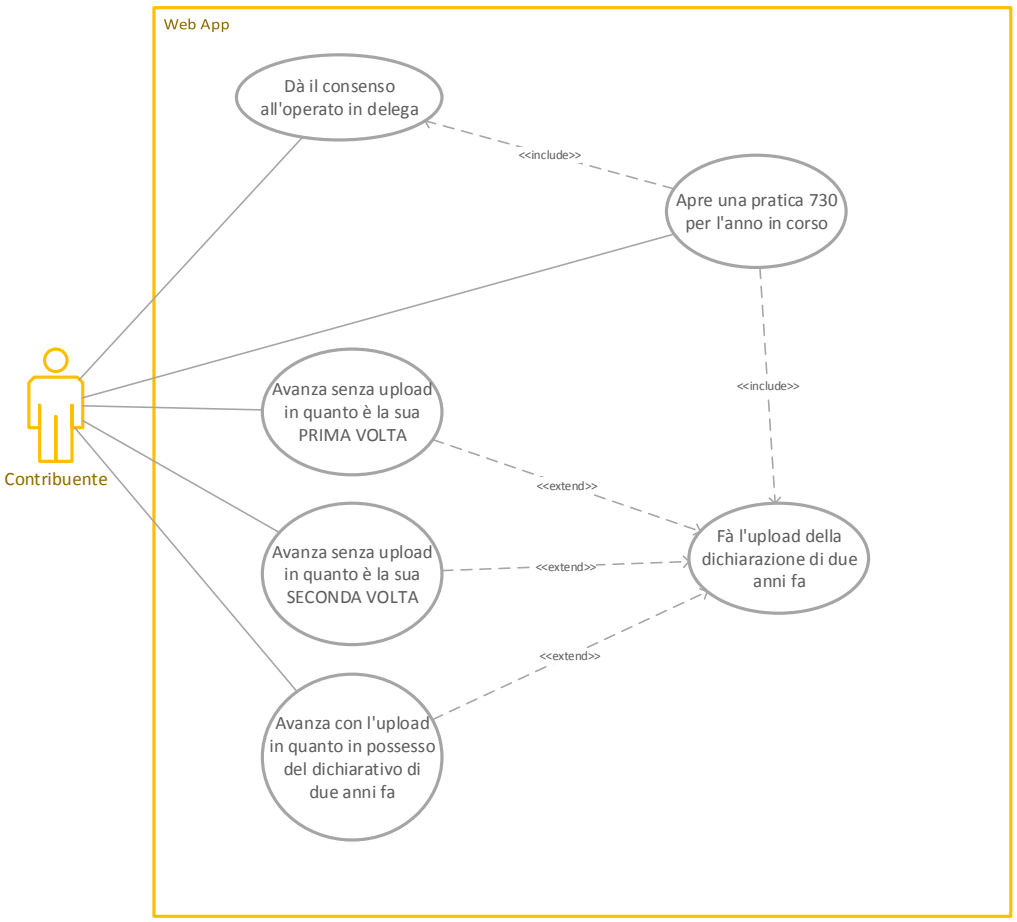

Figura 2.2: Per l'apertura della pratica 730 serve dare il consenso all'operato in delega sui portali del fisco ed uploadare il dichiarativo di due anni prima (quando disponibile).

2.1.5.3 Fase di input (Frontend)

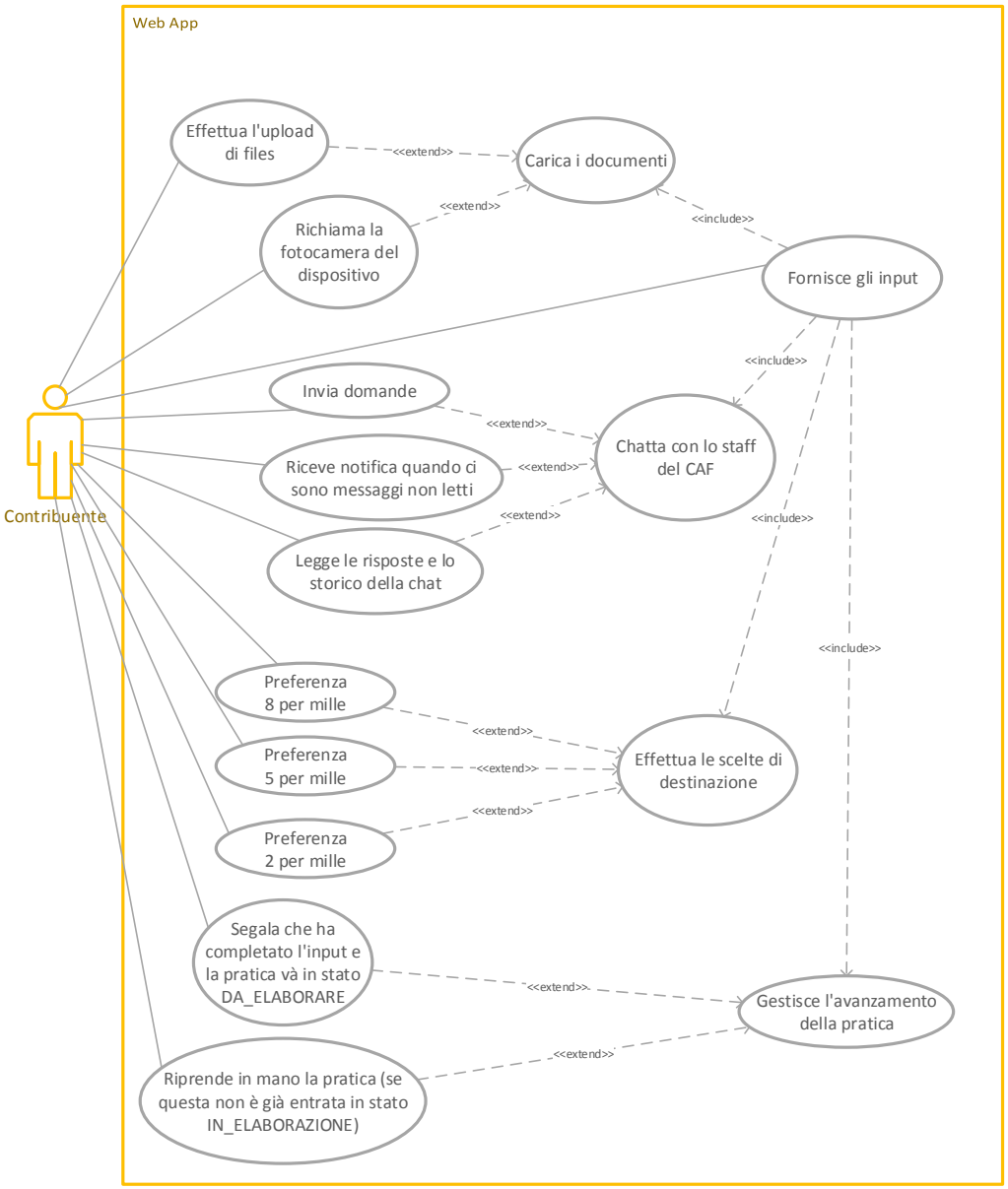

Figura 2.3: Il contribuente invia i documenti utili per l'elaborazione della sua pratica ed esprime le scelte di destinazione. Quando ha terminato segnala il fatto portando la pratica 'in attesa di essere elaborata'.

2.1.5.4 Fase di input (Backend)

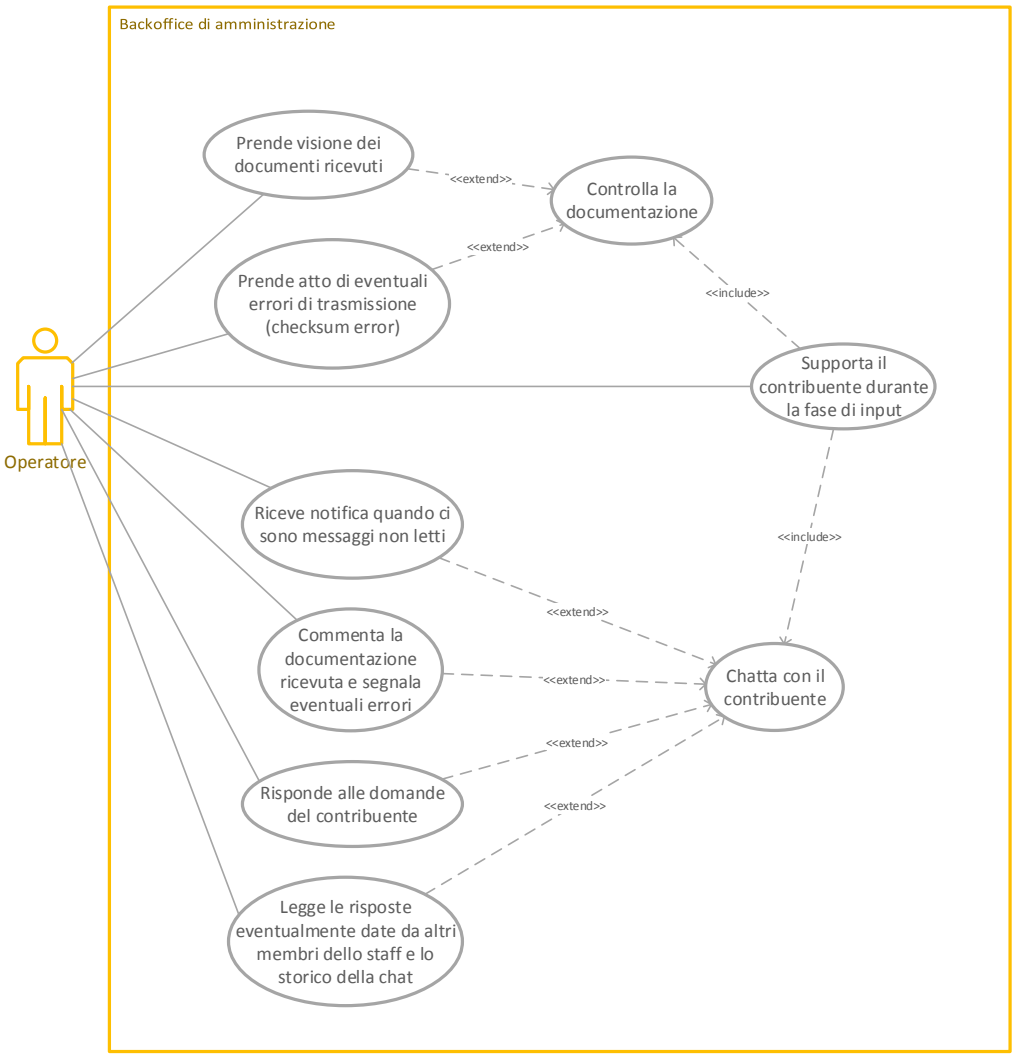

Figura 2.4: L'operatore visiona la documentazione e dà supporto tramite la chat segnalando eventuali errori di ricezione o carenze nella documentazione.

2.1.5.5 Fase di elaborazione

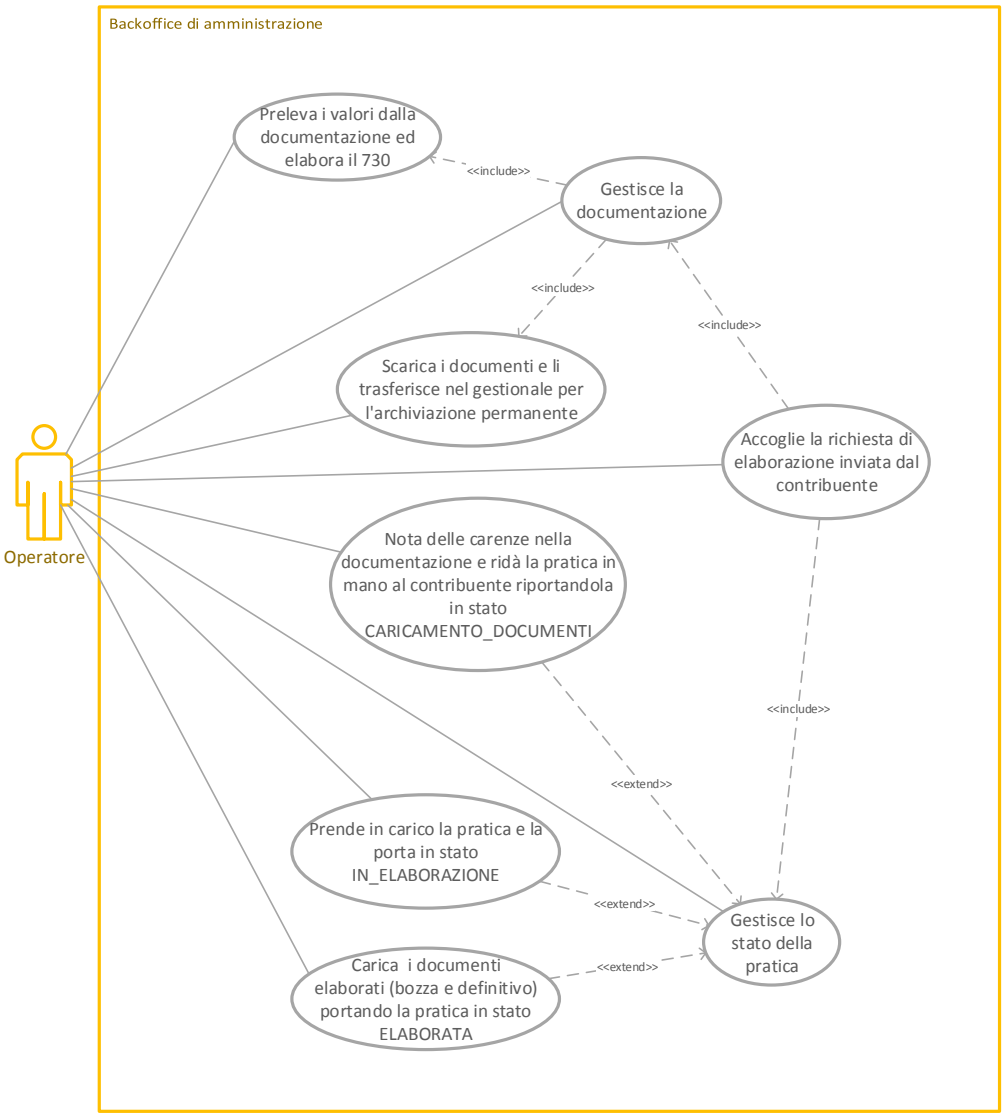

Figura 2.5: L'operatore prende in carico la pratica. Durante questa fase il contribuente non potrà più caricare documenti o cambiare le scelte fatte. Quando l'elaborato sarà pronto, verrà caricato nel sistema nella forma di una bozza e di un documento definitivo. La pratica a questo punto andrà in stato ELABORATA ed il contribuente potrà proseguire con le procedure conclusive.

2.1.5.6 Fase conclusiva

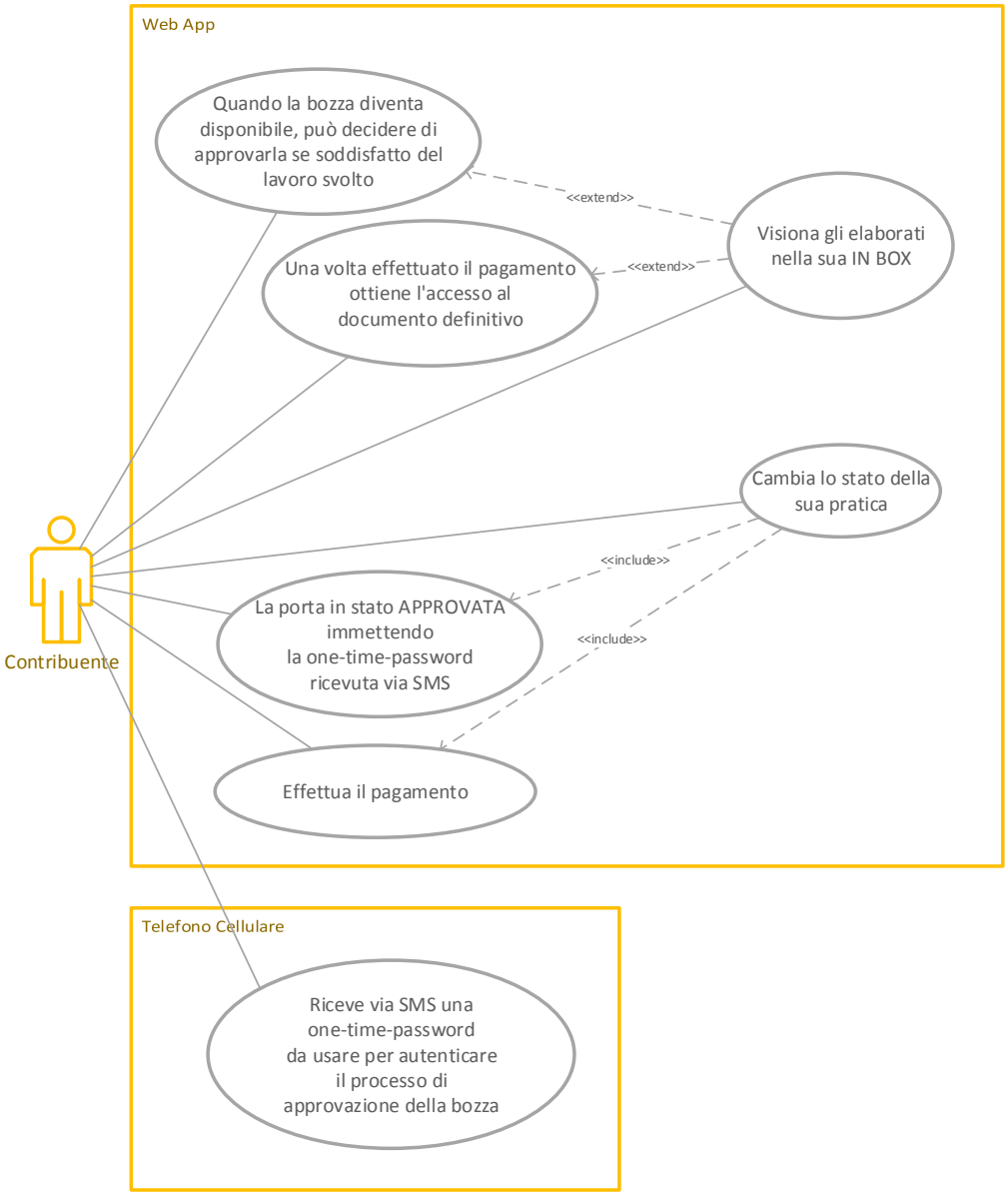

Figura 2.6: Il contribuente vede comparire nella sua IN BOX il documento bozza. Può visionarlo ed approvarlo inserendo la password che ha ricevuto via SMS. A questo punto potrà procedere con il pagamento. L'operatore all'avvicinarsi delle scadenze di legge provvederà a trasmettere il dichiarativo con suo apposito sistema in maniera trasparente al contribuente.

2.1.6 Diagramma degli stati

2.1.6.1 Registrazione a il 730 online

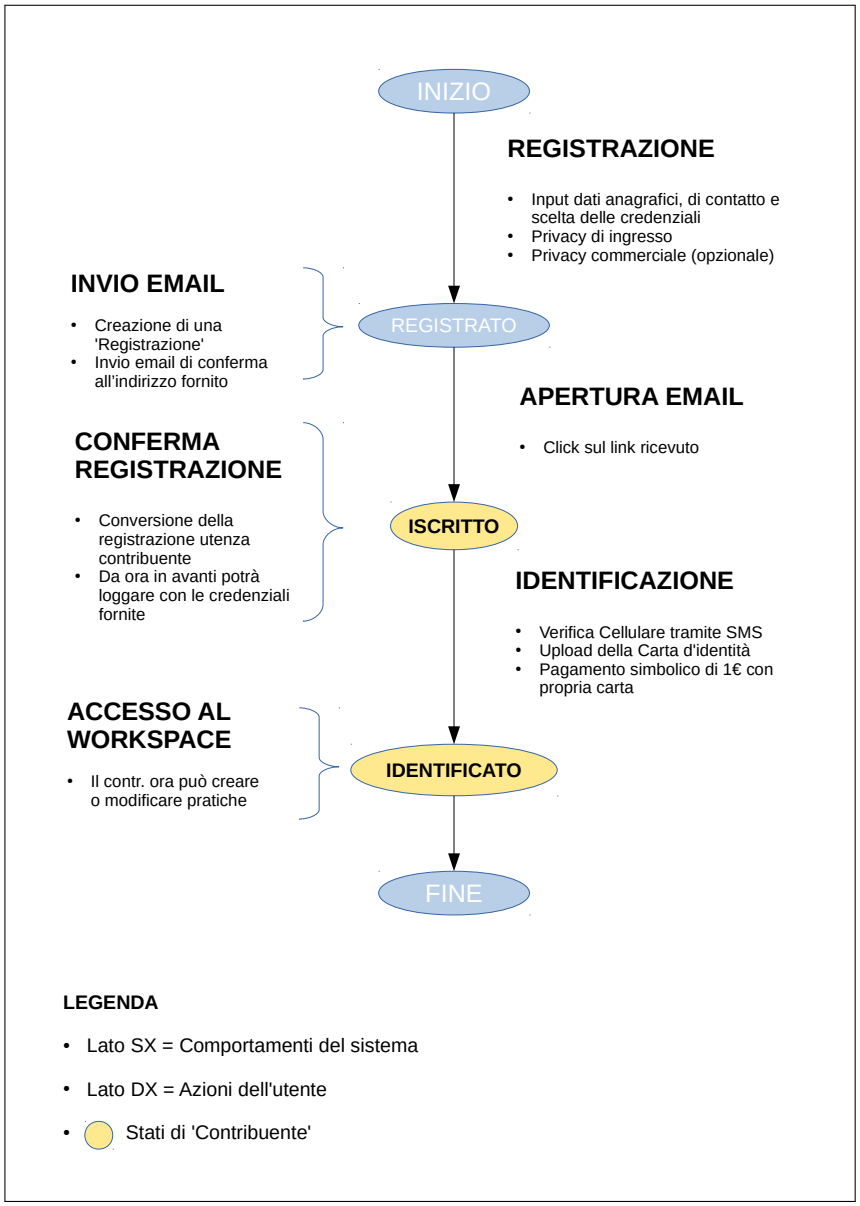

Figura 2.7: Si evidenzia il passaggio in cui una registrazione effettuata da un *Guest* viene confermata riconducendola ad un'utenza abilitata al login. Questa utenza viene ulteriormente verificata prima di ottenere l'accesso completo alle funzionalità del sistema.

2.1.6.2 Stati documentali della pratica 730

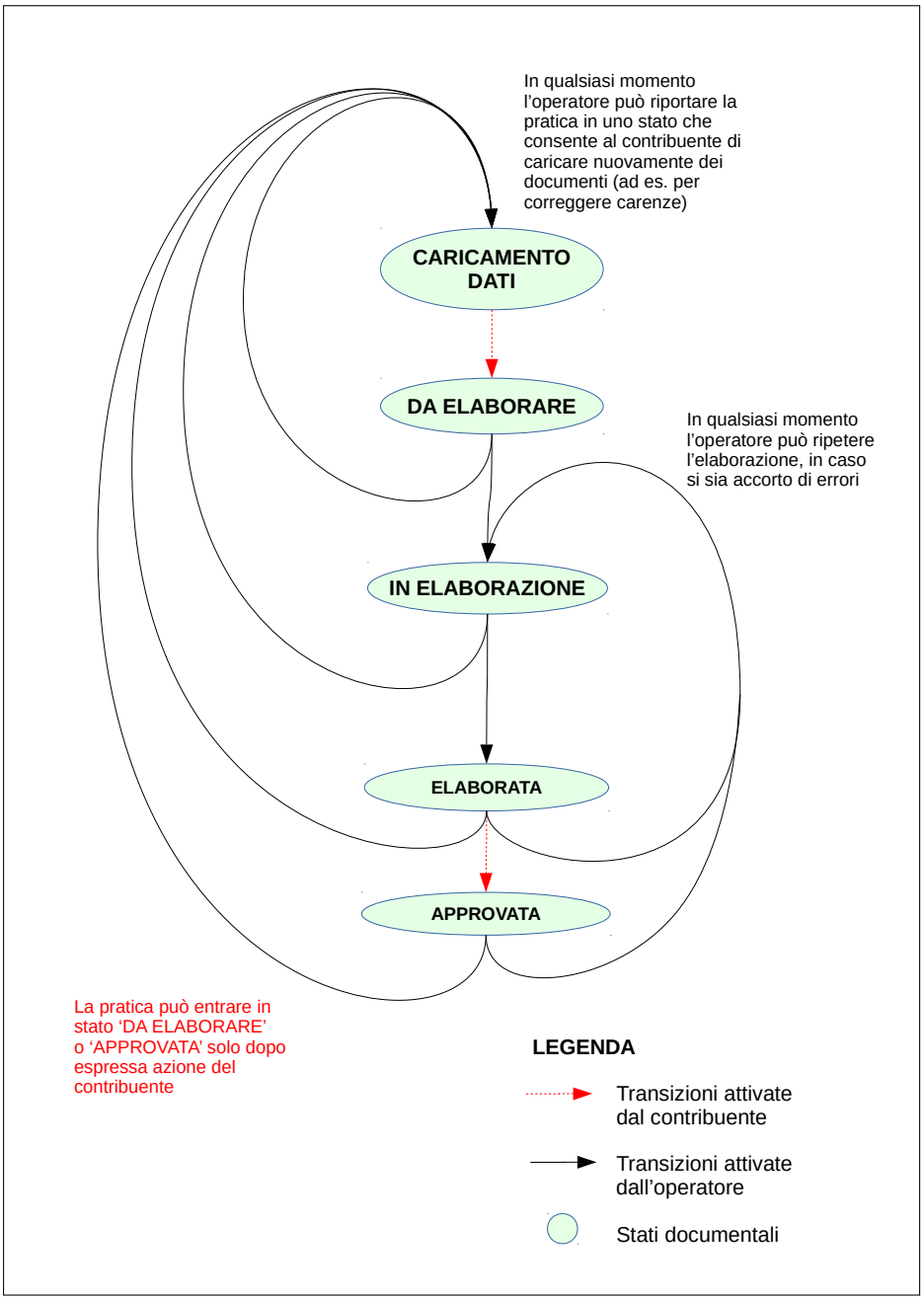

Figura 2.8: Una volta che la procedura di apertura della pratica 730 è stata completata, questa può essere gestita. Ogni fase della gestione comporta responsabilità diverse per il contribuente e per l'operatore.

2.2 Design e progettazione

2.2.1 Componenti architetturali del sistema

La funzionalità al centro del sistema è quella dello scambio di documenti tra contribuenti ed operatori. Serve dunque un logo dove questi siano archiviati (che indicheremo come **media storage**).

Questi files devono essere anche corredati da informazioni aggiuntive, necessarie per ricondurne la paternità e l'appartenenza alla rispettiva pratica. Oltre a queste informazioni strettamente documentali ce ne sono altre di vario tipo necessarie per il corretto svolgimento delle varie operazioni e per storicizzare tutto quanto avvenuto. Alcuni esempi di questi dati possono essere: l'elenco di utenti ed operatori, l'elenco delle pratiche ed i loro relativi stati di evoluzione, lo storico messaggi nelle chat, lo storico esiti dei pagamenti eccetera. Per questi motivi il sistema doveva prevedere (in aggiunta al sistema di archiviazione 'media') un sistema di archiviazione di informazioni strutturate (che indicheremo come **database**).

L'alto livello di reattività che ci siamo prefissati per donare al cliente finale una piacevole esperienza d'uso ha imposto che molta della elaborazione *front-end* dovesse avvenire direttamente sul suo dispositivo. Questo può essere ottenuto solo con una applicazione web *single page* o con una app nativa. Vista l'eterogeneità dei dispositivi e la necessità di avere un'unica *fonte di verità*, era impensabile scegliere di usare il dispositivo anche come supporto di archiviazione, quindi il sistema doveva per forza di cose avere una connotazione distribuita **client-server**.

Le logiche in opera sul server dovevano assolvere come minimo ai seguenti scopi:

- Centralizzatore le informazioni (files + dati).
- Mediatore l'interazione con i provider di servizi esterni (gateway di pagamento bancario e fornitore di invii SMS)
- Fornire un'interfaccia autenticata di accesso ai dati (API), necessaria per supportare i client.

Và ricordato che per via dei **requisiti implementativi imposti dalla direzione**, occorreva *ovunque possibile* usare tecnologie e metodiche già note, senza introdurre particolari novità. Dal momento che il know how aziendale era basato sull'uso del linguaggio PHP e del framework Yii, questo requisito ha avuto un particolare impatto nel design della componente lato server del sistema, che di fatto è stata pensata sin dall'inizio per essere implementabile in questo modo.

Il framework Yii offre buoni strumenti per lavorare con il file system e con basi dati relazionali, ma le classi che mette a disposizione sono istanziabili solo nel contesto di una *applicazione Yii*. Nell'ottica di sfruttare al massimo il framework, tutti i servizi offerti

dal server elencati in precedenza, sono stati sviluppati come parti di un unico applicativo Yii.

Il database è stato trattato con gli strumenti di modeling nativi offerti dal framework. Il servizio a supporto delle richieste client è stato costruita come un *modulo api* integrato nella logica MVC dell'applicativo. Allo stesso modo il backoffice di amministrazione è stato realizzato con una soluzione web tradizionale (1.3.2.1) integrando un *modulo dashboard* della logica dell'app lato server.

Và detto che al di là delle scelte imposte, quella di eseguire l'interfaccia di amministrazione in stretta vicinanza con i dati è buona dal punto di vista delle prestazioni, dal momento che il livello di interrogazioni fatte dagli operatori è alto e la quantità di banda necessaria per accedere ai documenti lo è altrettanto.

Il nome del sistema che eroga tutti i servizi fin'ora descritti è **platform.il730.online**

Fino ad ora l'unico client sviluppato è una Single Page Application ed al momento della discussione di questa tesi non sono previsti sviluppi di app native. Questa applicazione web necessita di un server HTTP per essere consegnata al browser dove andrà in esecuzione; considerando che è di fatto composta da sole risorse statiche, per essere servita necessita di un supporto tecnico essenziale. **Per essere più robusti a fronte di attacchi e più flessibili a fronte di cambiamenti futuri abbiamo scelto di introdurre un altro server** dedicato solo per questo scopo. Il suo nome è **app.il730.online**.

Questa separazione permette di sfruttare al meglio le risorse virtualizzate della infrastruttura sottostante (2.1.3) e viene evidenziata in questa trattazione preliminare in quanto ha implicazioni sulle fasi di sviluppo a valle.

Una applicazione web che viene servita da un server (con un certo nome 'A') e che effettua chiamate asincrone ad un altro server che risponde ad un nome diverso (es. 'B') ricade in certe politiche di sicurezza attuate dal browser. Affinché la comunicazione possa avvenire con successo, occorre che l'API interrogata sul server destinatario implementi un meccanismo detto CORS (1.4.3).

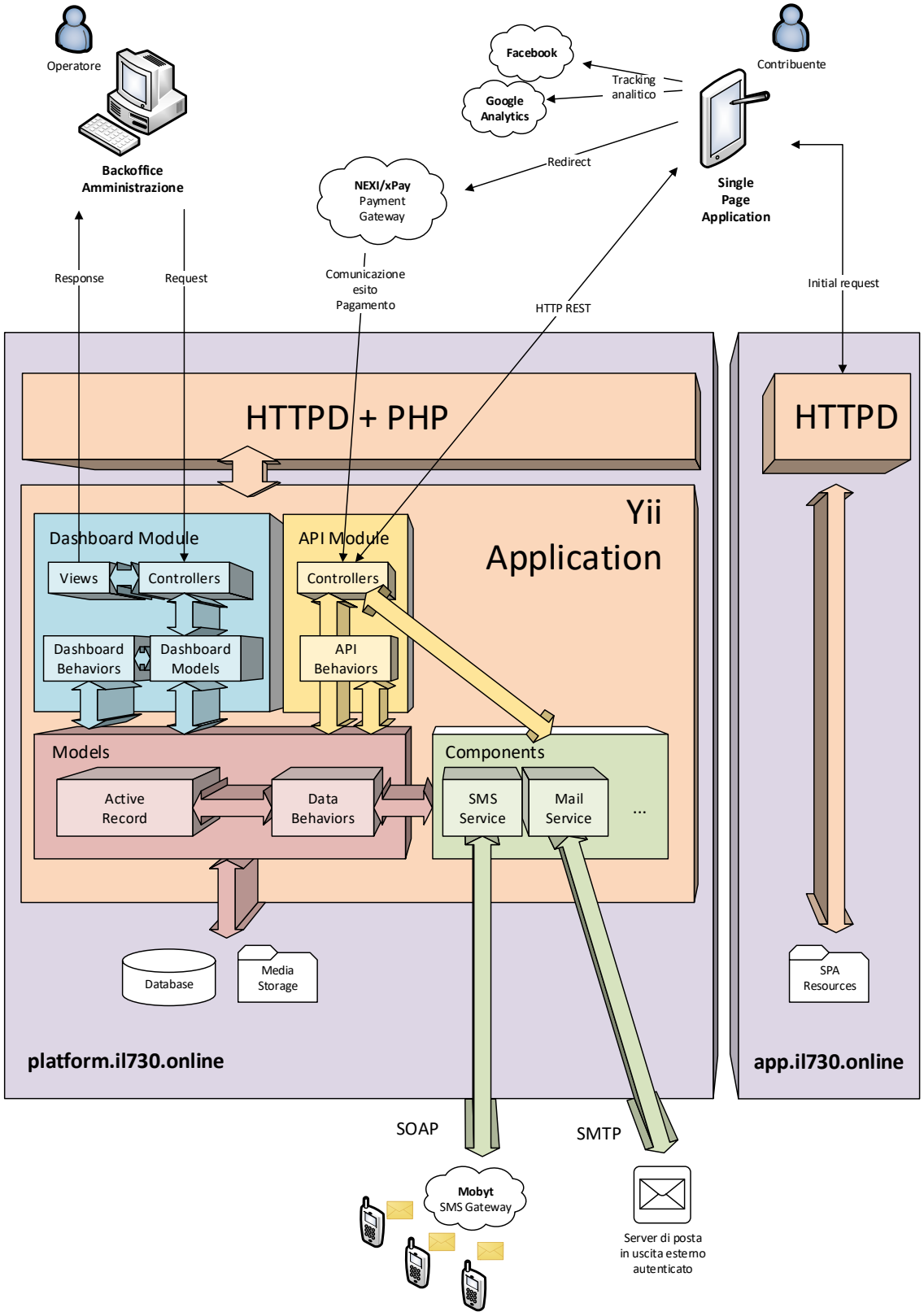

Figura 2.9: Architettura generale del sistema con evidenziati i flussi di interazione tra clients e servers.

2.2.2 Design architetturale

In questa sezione seguirà una trattazione sui vari componenti che sono stati individuati per i vari sistemi.

2.2.2.1 Sistema server 'app.il730.online'

Questo sistema ha lo scopo di **servire ai browser dei contribuenti il bundle di risorse statiche html, javascript e css che compongono la Single Page Application per loro progettata**.

La genesi di questo sistema deriva da una riflessione sul requisito di **efficienza e robustezza**.

Dal momento che tutta l'elaborazione della SPA avviene sul client, non servono risorse sul server (se non per servire la prima request con la quale l'applicazione viene scaricata sul client). Inoltre non è stato fatto nulla di particolare per prevenire il caching lato browser delle risorse, quindi ad un successivo approdo dello stesso utente c'è una grande probabilità per cui la mole di dati trasferiti sia veramente esigua.

Dal momento che l'infrastruttura hardware del cliente è virtualizzata, il fatto di prevedere sin dall'inizio una separazione a livello sistemistico permette una **migliore allocazione delle risorse**.

L'esperienza aziendale inoltre ci insegna che un form di login online, prima o poi viene preso di mira da attacchi di forza bruta operati da bot automatici. In certi casi l'accanimento di questi attacchi è tale da richiedere un impiego di risorse sul server non trascurabile.

Per fronteggiare queste situazioni esistono soluzioni standard di mercato (quali ad esempio *Cloud Flare*[9]) che operano ad un livello molto alto dello stack Internet (tipicamente a livello del servizio DNS). Il fatto di prevedere sin dalle prime fasi di design l'adozione di un *nome* specifico, permette di avere flessibilità in futuro a fronte di una eventuale necessità di questo tipo.

Per un altro grosso progetto realizzato in azienda, capitò che per motivi di performance il cliente chiedesse in corso d'opera di portare fisicamente il server che ospitava i dati ed il backoffice di amministrazione dentro la sua intranet ma senza inoltrare le richieste fatte dai clienti dalla rete pubblica. Il sistema in oggetto non era progettato per una richiesta di quel tipo e richiese un particolare setup di rete di non semplice gestione. Sebbene non avessimo per il progetto de IL 730 ONLINE un requisito del genere, il fatto di separare gli ambiti già a livello del nome di dominio, permette un alto livello di flessibilità che potrebbe fronteggiare con semplicità anche una situazione come quella prima descritta.

[9] *Cloudflare Anti DDoS* https://www.cloudflare.com/it-it/ddos/

Il sistema che serve l'app ha un design del tutto banale: trattasi sostanzialmente di un server HTTP configurato per mappare le request per il dominio `app.il730.online` su una directory contenente i codici lato client dell'applicazione.

Il fatto che la SPA sia servita da questo nome di dominio e faccia interrogazioni ad un server rispondente ad un diverso nome di dominio, implica in fase di implementazione dell'API la gestione delle così dette *richieste di preflight* e verrà discussa nel capitolo 2.3.2.4.

2.2.2.2 Sistema server 'platform.il730.online'

La progettazione architetturale ha visto l'individuazione dei seguenti sottosistemi:

Applicativo Yii Opera tutta la gestione delle informazioni modellando le strutture dati sottostanti (files e tabelle SQL).

Le request HTTP emesse dai vari client vengono gestite dal framework attivando un controller in base a regole di mappatura configurabili. Questo *routing* tiene conto anche della strutturazione dell'applicativo in moduli. Infatti quando una richiesta interessa un controller gestito da un modulo anche quest'ultimo viene attivato.

Questo meccanismo è stato sfruttato per **supportare diversi tipi di client**. I possibili url di richiesta della nostra interfaccia sono stati segmentati in famiglie in modo che ad ognuna corrispondesse un modulo. A livello del modulo sono state implementate le logiche comuni a tutti controller in quell'area di gestione.

La divisione principale prevede che gli url che hanno come radice `dashboard` siano ricondotti al modulo responsabile di offrire il backoffice di amministrazione, mentre quelli che hanno `api` al modulo che supporta l'interazione con la Single Page Application e con altri client esterni come il gateway di pagamento (quando contatta il nostro server per notificare l'avvenuto pagamento).

Il backoffice non deve essere raggiungibile dai contribuenti quindi a livello del modulo dashboard viene fatto un controllo di questo tipo.

Certe operazioni sui dati sono ammissibili se fatte da un operatore ma non da un contribuente quali ad esempio l'impostazione del prezzo della pratica. A mio avviso inserire questo tipo di logiche 'di guardia' direttamente dentro alle strutture dati non è saggio, perché creerebbe un codice molto difficile da leggere (di logiche di questo tipo ce ne sono decine) e creerebbe una dipendenza *sull'attore* nelle entità più fondamentizie del sistema. La soluzione progettuale adottata è quella dei *behaviors*. Ogni ambito operativo (modulo) al momento della sua attivazione decora le strutture dati sensibili con delle logiche reattive ai loro eventi (es. 'pre salvataggio'). All'accadere dell'evento la logica del behavior si attiva e può decidere se bloccare l'azione in corso (es. salvataggio) o se apportare una correzione ai dati o se 'approvare'.

Questo approccio è risultato così flessibile ed espressivo che è stato usato per gestire tutte le problematiche di assegnamento delle strutture con valori di default, nell'implementazione delle logiche annuali e nella realizzazione di logiche di guardia per difendere da salti di stato proibiti.

Servizio esterno SMS Il provider scelto per l'invio degli SMS è Mobyt. Una volta registrato un account e caricato del credito tramite il loro apposito portale è possibile spedire SMS utilizzando una delle API che mettono a disposizione. Noi abbiamo scelto quella SOAP perché avrebbe consentito un guadagno nella scrittura del client (usando tool di processamento automatico del loro file WSDL). Da qualche release di PHP a questa parte, la funzionalità SOAP è stata estirpata dal core ed è disponibile solo come modulo separato. Per il funzionamento del nostro client occorre quindi provisionare ogni sistema (sviluppo, testing, staging, produzione) installando questo modulo di PHP.

Servizio esterno Mailer Abbiamo deciso di notificare le novità ai contribuenti (presenza di messaggi non letti nella chat, disponibilità dell'elaborato nella inbox) attraverso normali email.

Uno dei requisiti fondamentali del sistema era quello di essere usabile da qualsiasi dispositivo: ad esempio è promosso il fatto che un contribuente si iscriva da pc per poi caricare i documenti dallo smartphone, sfruttando la fotocamera.

In quest'ottica è importante avere **omogeneità**. La tecnologia della email è disponibile su qualsiasi dispositivo e la quasi totalità degli utenti tiene la casella di posta correttamente configurata anche sul suo smartphone o tablet.

Soluzioni alternative come le *notifiche push* offerte dalla tecnologia cloud di Google sono a pagamento, mentre le notifiche in-browser sebbene abbiano un supporto tecnologico abbastanza decente[10] per essere implementate richiedono un approccio progettuale che tende a quello delle Progressive Web App (1.3.2.3) e non abbiamo voluto metterlo in campo in queste prime release.

PHP permette la spedizione di email anche a livello software sfruttando API native ma abbiamo scelto di non intraprendere questa strada. L'adozione di un server SMTP autenticato aumenta la probabilità di superare i filtri anti spam ed il fatto di usare un protocollo intermediario tra l'applicazione ed il servizio, permette di disaccoppiarla da quest'ultimo. Come sarà descritto nel capitolo sul testing, durante lo sviluppo abbiamo impiegato un server di posta in uscita speciale che permette di verificare il funzionamento ma senza effettuare spedizioni sulla rete pubblica.

Sono questi i motivi che ci hanno spinto all'adozione di un servizio a livello di sistema.

[10] *Can I Use 'Web Notifications'* https://caniuse.com/#search=notification

2.2.2.3 Sistema client Single Page Application

Supporto al routing Il cuore di una SPA è la sua macchina a stati finiti.

In alcuni scenari l'utente abbandona temporaneamente l'applicazione per poi ritornarvi in uno **stato specifico**: questo avviene ad esempio durante la registrazione e durante i pagamenti.

L'unico modo per indicare ad un browser una destinazione è tramite l'URL ma sappiamo che questo non ha senso in una SPA, in quanto l'unico URL fisicamente servito dal server è quello corrispondente alla `index.html` che incorpora l'intera applicazione. Per supportare questo tipo di navigazione **serve dunque collaborazione da parte del server**. Tutte le richieste diverse dall'entry point dell'applicazione non devono essere respinte con un errore `404 Not Found`, ma **redirezionate** verso l'index in modo che sia poi l'app a mappare lo stato corrispondente all'URL richiesto.

Uno stato può dunque diventare attivo o *via software*, in risposta ad un azione dell'utente sull'interfaccia grafica, o *via navigazione* cioè quando un URL di quel dominio viene catturato dalle regole di redirect lato server e gestito dal *router* dell'app riconducendolo ad uno stato *noto*.

Considerazioni sul life-cycle dell'applicazione Particolare enfasi và messa proprio sulla parola '*noto*' usata nel paragrafo precedente.

Una qualsivoglia single page application prima di processare la request deve avere *noti* quali sono gli stati disponibili e l'intero grafo delle connessioni.

Questa considerazione è molto importante perché di fatto implica la divisione del ciclo di vita dell'applicazione in **due fasi ben distinte**: una fase di **config** in cui tutti gli stati del sistema vengono enumerati e contribuiscono a formare una *coscienza del router* ed una fase di **run** in cui la richiesta viene ricondotta ad una serie di transizioni elementari che permettono di raggiungere lo stato in oggetto partendo da quello *root*.

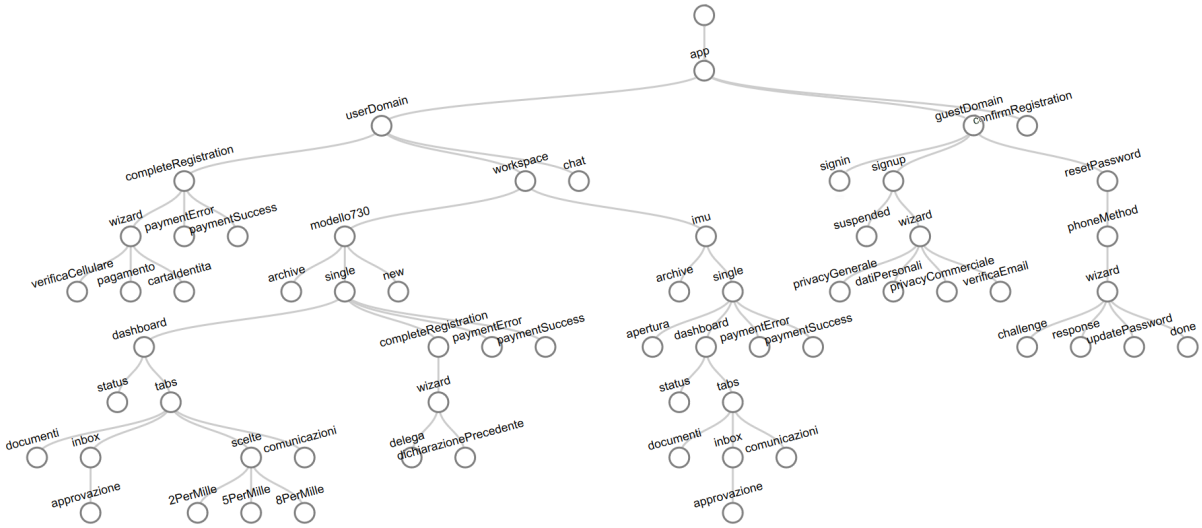

Figura 2.10: Il grafo stati dell'applicazione al momento della stesura di questo documento (mostra infatti alcune features implementate in seguito e non espressamente trattate in questa tesi, come la gestione dei tributi locali IMU).

Data flow Ogni transizione comporta l'abbandono di uno stato di partenza e l'ingresso in uno stato di destinazione. Certi tipi di logiche del sistema possono essere implementate perfettamente sotto forma di gestori degli eventi *onEnter* ed *onExit* emessi dagli stati coinvolti dalla transizione.

Nella fase di ingresso di uno stato si possono eseguire le logiche di guardia che verificano che in quel momento si disponga dei requisiti per l'accesso a quella particolare sottosezione del grafo (si noti nella figura 2.10 la grande dicotomia tra *guest domain* a destra ed *user domain* a sinistra).

Un'altro uso principe della meccanica ad eventi sull'ingresso è quella di *risolvere* le entità necessarie per parametrizzare i componenti da attivare per quello stato. Ad esempio prima di entrare nella dashboard di gestione di una pratica occorre aver fatto un interrogazione all'API per ottenere una rappresentazione della risorsa remota. Se per qualche motivo la richiesta non dovesse andare a buon fine non ci sono gli elementi per approdare in quello stato e la transizione deve essere abortita.

La meccanica onExit invece è utile invece per annullare operazioni in corso o per eseguire il rilascio di risorse impegnate.

MVC Una volta che la transizione è completata il sistema provvede ad instanziare le *views* previste per quello stato e ad incastonarle nella pagina. A questo punto vengono istanziati i *controllers* associati a quelle viste e parametrizzati di tutte le loro dipendenze (models e services) tramite il meccanismo della *Dependency Injection* (1.1.1.1).

L'applicazione è ora pronta per gestire le interazioni dell'utente ed effettuare nuove transizioni di stato.

Stati Abstract Non è richiesto che tutti gli stati siano *approdabili*. La possibilità di incorporare la logica del sistema nelle transazioni è così utile che spesso conviene introdurre stati 'di passaggio' appositamente per questo scopo.

Lo stato *user domain* citato in precedenza è proprio uno di questi esempi, in quanto non è direttamente navigabile ma ha lo scopo di centralizzare per tutta l'area di sinistra il controllo sul fatto che l'utente sia autenticato.

Questo tipo di stati intermediari sono detti *stati astratti*.

Per riassumere, durante la fase di configurazione bisogna indicare per ogni stato le seguenti informazioni:

- Se è uno stato astratto o no. In caso contrario qual'è la porzione di URL della richiesta da ricondurre ad esso.
- Quali sono i *bundles* di vista + controller da istanziare all'attivazione dello stato e dove essi devono essere inseriti nella pagina.
- Quali sono i dati da risolvere in entrata allo stato, prima di procedere all'istanziamento dei componenti.
- Eventuali *handlers* per gli eventi onEnter ed onExit.

Gestione della history Nel mondo Android il tasto `back` è molto importante per la navigazione ed ha il significato di *'torna alla videata precedente'*.

Nel contesto di una SPA l'aggiornamento della videata non comporta una vera e propria navigazione, quindi ciò che ne deriva è che la storia del browser sia naturalmente sempre vuota.

In questa situazione alla prima pressione del tasto back avverrebbe la chiusura del browser! In quanto di fatto la sua apertura è stata l'unica azione nota e registrata.

Ciò che deve fare una SPA ben fatta è aggiungere uno step alla **browser history** ad ogni nuova transazione. In questo modo la storia di navigazione dell'utente viene popolata e quando esso si sposta avanti o indietro, l'applicazione può catturare gli eventi emessi dal browser per ripristinare lo stato in cui si trovava in precedenza.

Autenticazione Per questa applicazione abbiamo cercato di fare scelte architetturali tali da sviluppare una soluzione facilmente portabile in futuro verso altre piattaforme.

Lo sbocco più probabile era quello di convertire un giorno la SPA in applicazione ibrida.

In quell'ambito operativo il meccanismo di autenticazione tradizionale del web (cioè quello basato su cookie lato browser e sessioni lato server) non è supportato.

Come alternativa abbiamo scelto un meccanismo di autenticazione stateless basato su token (1.4.2.2) ed il suo funzionamento è il seguente:

1. L'utente immette le credenziali (email e password) ed effettua il login.
2. L'applicazione effettua una chiamata all'API lato server dove avviene la verifica. L'identità del soggetto che è dietro a queste credenziali viene imbustata in un `JSON Web Token` che viene ritornato al client.
3. Il client ora conserva questo token *opaco* in una qualche forma di persistenza, come un file o la *storage*[11] del browser.
4. Per tutte le future richieste all'API invia tramite un header HTTP il token memorizzato.
5. Questo viene ricevuto lato server e decodificato per recuperare la sessione utente senza coinvolgere nessuna risorsa del server.
6. A questo punto la richiesta API viene gestita sapendo che la richiesta impersonava un certo specifico utente.

Il sistema è particolarmente adatto per le applicazioni su larga scala perché il fatto che non esista nessuna informazione di sessione memorizzata sul server permette ad esempio di attuare politiche di balancing su più server senza problemi.

[11] *Browser Storage* https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API

Un'esempio concreto Ipotizziamo di voler inviare una mail di sollecito agli utenti che hanno una pratica bloccata perché non hanno ancora provveduto ad esprimere le loro scelte di destinazione. Questa dovrà contenere una *call to action* per portare l'utente direttamente nell'applicazione allo stato interessato.

Attraverso una email non è possibile *programmare* la navigazione di una SPA: l'unica cosa che si può fare è inserire un link. Questo deve essere abbastanza espressivo da permettere al router della SPA di trarre tutte le informazioni necessarie per portare l'utente sullo stato desiderato.

L'URL per questo esempio è `/modello730/2018/scelte/` ed ora verranno elencate le azioni che si susseguiranno una volta avviata la navigazione verso di esso.

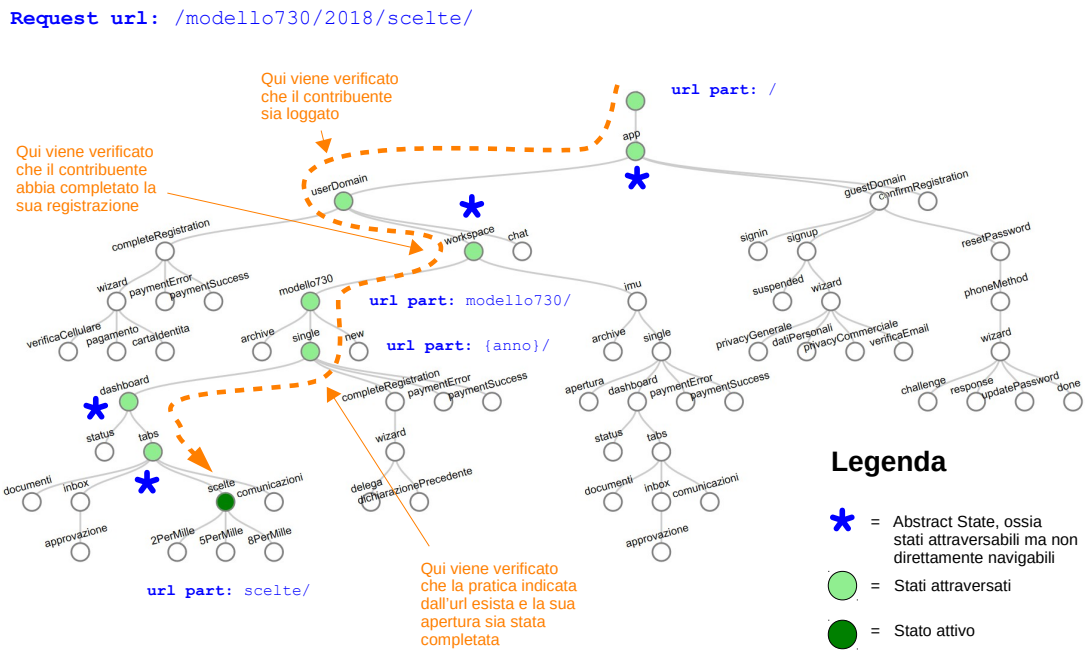

Figura 2.11: Grafo in cui sono evidenziate le attività coinvolte nell'esempio.

1. Il sistema esegue la fase di config e tutti gli stati si *registrano* in modo che il router possa considerarli durante la navigazione.
2. Il sistema entra nella fase di run. Il router reagisce a questo evento e comincia a parsare l'URL fino a trovare una catena di stati che corrisponde alla richiesta effettuata. In figura è evidenziata col tratteggio arancione.
3. Il router attiva lo stato astratto `app`: in questa fase viene impostato il layout di base dell'applicazione.

4. Il router attiva lo stato astratto `userDomain`: in questa fase si controlla che nella storage locale del browser ci sia un token di autenticazione ottenuto in una precedente operazione di login. Supponiamo che sia così.
5. Il router attiva lo stato astratto `workspace`: in questa fase si controlla che l'utente abbia completato la sua registrazione validando la sua identità, in modo da concedergli pieno accesso ai servizi dell'applicazione.
6. Il router attiva lo stato `modello730` che ha il solo scopo di disambiguare l'annualità seguente e poi il suo sottostato `single`. In questa fase viene prelevato il componente dell'URL '2018' e fatta una interrogazione all'API (autenticata con il token) per ottenere una rappresentazione della pratica 2018 del soggetto loggato.
7. Il router attiva gli stati `dashboard` e `tabs` che hanno lo scopo di impostare l'interfaccia grafica di quella pratica.
8. Infine viene attivato lo stato `scelte` ed il componente che permette all'utente di esprimere la sua volontà viene istanziato ed inserito nella view.

Integrazione con servizi di analitica Per supportare i piani di investimento online del committente abbiamo dovuto fare in modo che l'applicazione collezionasse statistiche di navigazione per profilare il comportamento (in forma anonima ed aggregata) degli utenti.

Per la piattaforma Facebook questo si ottiene inserendo nella pagina un *pixel*[12] ossia un codice particolare che si attiva durante la sessione di navigazione dell'utente e provvede in background a collezionare alcune informazioni sul suo comportamento.

Per la piattaforma di Google il funzionamento è simile. Allo stesso modo il gestore dell'account statistiche deve generare lo *snippett*[13] di codice di tracciamento. Questo deve essere incorporato nell'applicazione ma questa volta serve fare uno sforzo ulteriore. La libreria di Google Analytics è in grado di identificare autonomamente le *page views* in base alle navigazioni che avvengono nel browser. Siccome la SPA non le prevede, queste devono essere segnalate manualmente emettendo un evento analitico ad ogni transizione del router.

[12] *Facebook Pixel* https://www.facebook.com/business/help/553691765029382
[13] *Google Analytics Tracking Code* https://support.google.com/analytics/answer/6086097?hl=it

2.2.3 Design dell'interfaccia

In questo capitolo tratteremo solo la progettazione del frontend per i clienti finali, in quanto il backoffice di amministrazione ha un look essenziale pensato per addetti tecnici.

L'interfaccia dell'app è stata progettata tenendo conto del vasto parco di utenti a cui doveva rivolgersi, con particolare ottimizzazione per l'aspetto *responsive* del layout.

Questo doveva infatti adattarsi ad un un range di risoluzioni molto vasto, partendo da quella di uno schermo HD da PC (1920px) fino ad arrivare a quella di uno smartphone di fascia bassa (280px, ossia di circa un settimo).

La volontà del committente era quella di far elaborare l'aspetto grafico da una loro agenzia di fiducia. Noi abbiamo provveduto a fare lo studio dell'interfaccia utente decidendo come organizzare i flussi applicativi e gli ingombri degli elementi. In seguito abbiamo elaborato dei mockup da fornire all'agenzia per elaborare il layout in maniera visualmente gradevole.

2.2.3.1 Considerazioni Generali

Vista l'esiguità dello spazio a disposizione abbiamo optato per spostare la **navigazione di primo livello** in una *sidebar* che rimane fissa sul lato sinistro per gli schermi desktop, mentre diventa a scomparsa per quelli mobile. Questo tipo di soluzione è nota in gergo come *drawer* (cassetto).

Una volta che l'utente ha scelto una voce di navigazione di primo livello dalla sidebar, questa collassa e lascia spazio ad una dashboard *fullscreen* dalla quale può articolarsi attraverso tutte le possibili operazioni previste per quell'area.

La **navigazione di secondo livello**, ossia quella che permette di muoversi tra le varie aree di gestione di quella dashboard è ottenuta con una vista a *tab*.

Ovunque era necessario chiedere un ulteriore input all'utente o dettagliare un informazione su sua richiesta abbiamo fatto uso estensivo di **finestre modali** *pop-up*.

Queste scelte hanno permesso di realizzare un interfaccia grafica che offrisse all'utente tutti i comandi a disposizione senza dover mai abbandonare la schermata, in modo che non servisse da parte sua un approccio esplorativo per scoprire le funzionalità del sistema.

Riscontriamo che il lavoro fatto dall'agenzia esterna per quanto riguarda lo stile visuale, la scelta della terna dei colori, dei font e delle spaziature, riprende vagamente il movimento del *Material Design*[14].

[14] *Material Design* https://material.io/design/

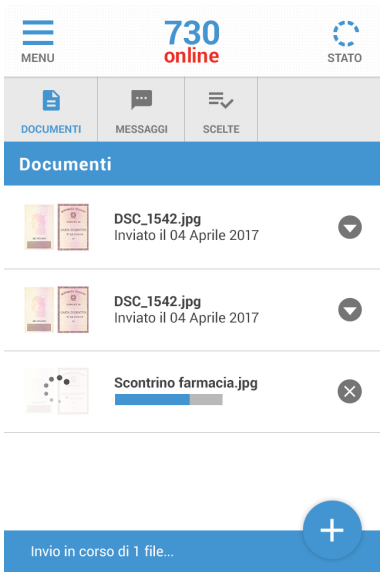

Figura 2.12: Layout mobile.

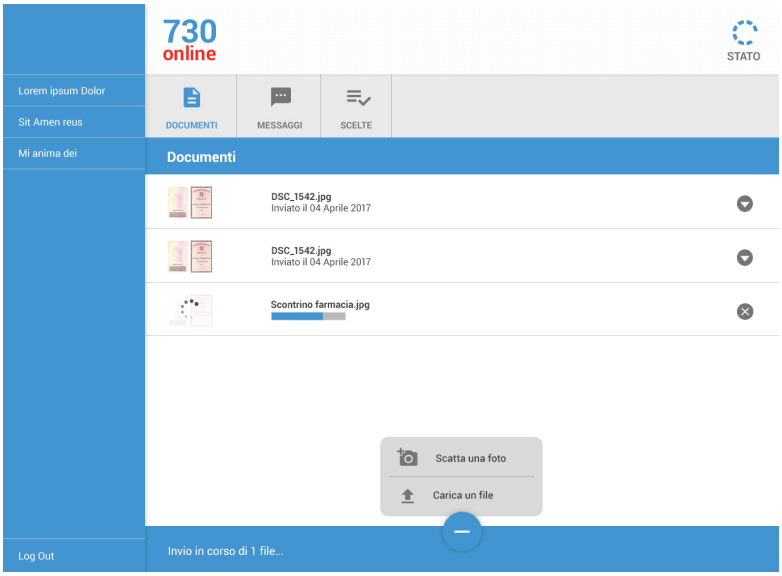

Figura 2.13: Layout desktop.

2.2.3.2 Fase di registrazione

Vista la laboriosità delle fasi di registrazione abbiamo pensato di non richiedere tutte le informazioni all'utente in un'unica soluzione, in quanto avremmo potuto spaventare i navigatori non preparati.

Abbiamo invece scelto di richiedere le informazioni un pò alla volta, con un'**interfaccia guidata step-by-step** che tipicamente viene chiamata *wizard*.

Il fatto di spezzare la procedura di acquisizione dei dati in più fasi permette all'utente di **interromperla per poi riprenderla** in un secondo momento e questo và a suo vantaggio.

Permette inoltre a noi di **tracciare i vari punti di abbandono**, in modo da ottimizzare quelli più significativi e di sviluppare eventualmente campagne di richiamo per recuperare quegli utenti che per noia hanno abbandonato la procedura ad un certo momento.

Le schermate coinvolte hanno un contenuto piuttosto ovvio (display delle informative privacy, form di raccolta dei dati anagrafici, form di input per pagamento con carta eccetera) pertanto non verranno dettagliate ulteriormente.

2.2.3.3 Dashboard Modello 730

Questa è l'interfaccia principale dalla quale l'utente può eseguire tutte le operazioni a lui consentite per la gestione della sua pratica.

Le operazioni disponibili (e quindi il loro numero) variano in base allo stato della pratica (ad esempio la possibilità di visionare la bozza diventa disponibile solo dopo che la pratica è stata elaborata).

Abbiamo voluto che il **layout rimanesse il più uniforme possibile tra uno stato e l'altro** in modo da non disorientare l'utente.

In particolare è stato fatto un lavoro di **ottimizzazione degli ingombri** per scongiurare l'evenienza di dover modificare il layout per fare spazio ad elementi UI aggiuntivi richiesti da certi particolari stati.

Le operazioni che può svolgere l'utente possono essere raggruppate in 4 categorie:

Operazioni di input Upload di documenti, scelte di destinazione.

Operazioni di output Visione della bozza, download del definitivo.

Chat Scrittura e lettura di messagi, ricezione di notifiche.

Controllo dello stato Presa visione dello stato attuale, invio della pratica per l'elaborazione, approvazione della bozza.

L'ultima voce dell'elenco sebbene sia comunque dipendente dallo stato della pratica deve essere sempre disponibile sullo schermo. Abbiamo deciso per questo motivo di escluderla della vista a tab dedicandogli l'angolo in alto a destra della videata come visibile nelle figure 2.12 e 2.13.

Tutte le altre aree di gestione saranno invece rese disponibili oppure no mostrando o oscurando il loro rispettivo tab.

A fondo pagina viene mostrata una **status bar** che ha il duplice scopo di mostrare un informazione all'utente e di ospitare i comandi contestuali per quella particolare tab.

I comandi sono sempre stati collocati sul lato destro in modo da essere facilmente raggiungibili con il pollice in ambito mobile.

Tab documenti Deve permettere all'utente di uploadare dei file selezionandoli dal suo archivio locale o attivando la fotocamera. Deve inoltre mostrare lo storico degli invii ed un feedback sull'esito della trasmissione (completato, annullato, corrotto).

Figura 2.14: Selezione del file.

Figura 2.15: Upload del file in corso.

La status bar del tab documenti è caratterizzata da un tasto +.
Alla sua pressione compare una *action list* che propone all'utente la scelta di un file dal suo dispositivo o l'attivazione della fotocamera. Il file scelto o lo scatto realizzato viene subito trasmesso e si può monitorare l'andamento complessivo degli upload dalle *progress bars* e dall'informazione di stato in fondo.

Chat Deve permettere all'utente di inviare messaggi al CAF e di vedere lo storico delle comunicazioni.

Figura 2.16: Tab chat.

Figura 2.17: Composizione messaggio.

La status bar della tab chat è caratterizzata da un pulsante di invio nuovo messaggio. Alla sua pressione compare un'interfaccia dove inserire il testo del messaggio ed un ulteriore pulsante per confermare l'invio.

Tab scelte Deve permettere all'utente di effettuare le scelte di destinazione per l'8, 5 e 2 per mille. Per ogni scelta deve essere possibile decidere di *non esprimere una preferenza*.

La scelta è da effettuarsi su un **numero chiuso di opzioni** ma l'elenco dei soggetti viene **deliberato di anno in anno**, quindi l'implementazione della UI non deve essere *hardcoded* ma deve popolarsi dinamicamente con l'elenco corrispondente a quello della normativa di riferimento per la pratica che si sta gestendo.

Per alcune *finalità destinatarie* del 5 per mille deve essere inoltre possibile indicare il **codice fiscale del soggetto beneficiario** disponendo di un campo aggiuntivo per l'eventuale inserimento di quest'ultimo.

Figura 2.18: Scelte.

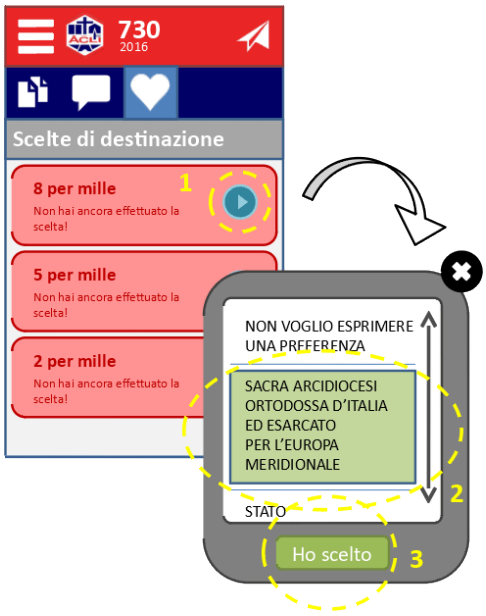

Figura 2.19: 8 per mille.

Figura 2.20: Update.

La tab scelte propone un elenco con una voce per ogni tipo di scelta da effettuare. É importante che questo evidenzi eventuali carenze nella compilazione in quanto la pratica non può essere sottomessa per l'elaborazione finché il contribuente non ha indicato tutte le sue volontà.

Al click su una voce compare a video un popup dove viene presentato in maniera grafica l'elenco dei possibili destinatari. L'utente può cliccare su una delle opzioni disponibili o su quella di *astensione*. In ogni caso la scelta deve essere ulteriormente confermata con la pressione di un pulsante: questo per evitare selezioni accidentali.

Alla chiusura del popup si può notare come l'elenco si sia aggiornato dando il feedback all'utente che ora sono rimaste solo 2 scelte da esprimere.

Nel processo di elaborazione del 730 cartaceo, tutti i campi del modello vengono compilati dall'operatore ad eccezione di quelli delle scelte. É infatti il contribuente a compilare quella parte del documento *apponendo la sua firma* sulle apposite caselle dei soggetti destinatari. Ipotizziamo pertanto che i clienti tradizionali conoscano il formato di quei moduli e siano abituati alla loro compilazione.
Per questo motivo abbiamo voluto emularli in modo da fornire un *look'n feel* familiare.
Sotto segue uno screenshot che mostra come è risultata l'implementazione dell'interfaccia grafica seguendo questi principi.

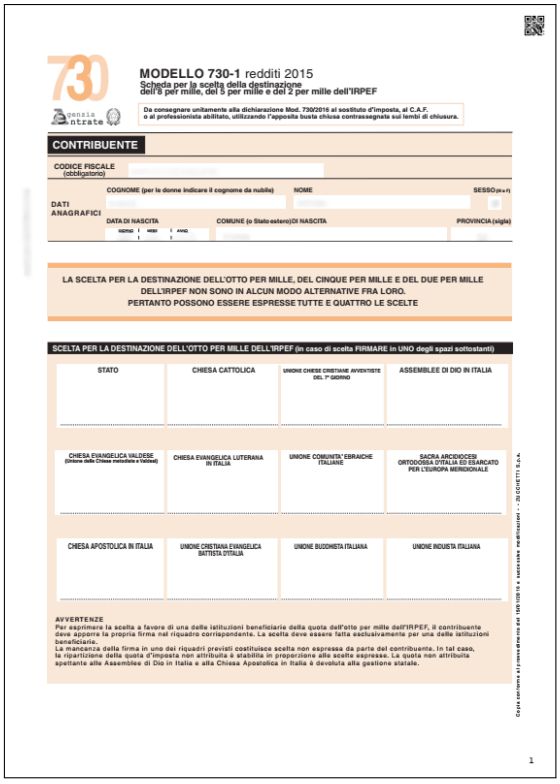

Figura 2.21: Modulo scelte 8 per mille.

Figura 2.22: UI di scelta dell'8 per millle con look'n feel analogo al modulo cartaceo.

Status Panel In ogni stato della pratica il contribuente deve poter indagare la situazione in modo da **sapere cosa deve fare o cosa deve aspettarsi**. Per questo scopo abbiamo deciso di predisporre uno *status panel* attivabile da un pulsante in alto a destra.

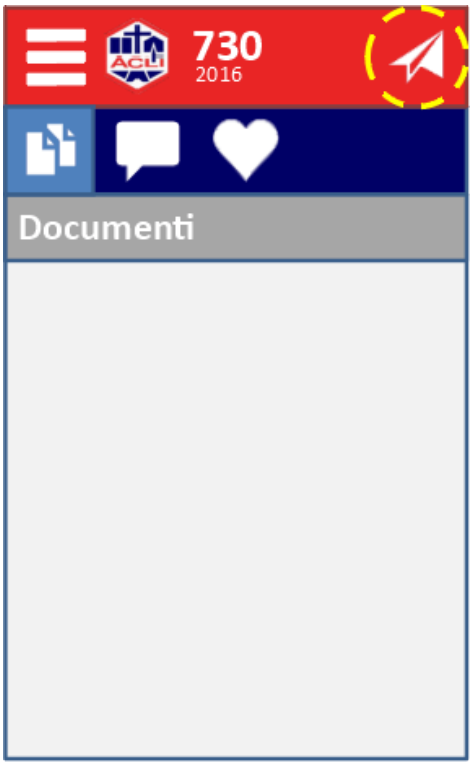

Figura 2.23: Ubicazione del bottone per mostrare lo status panel.

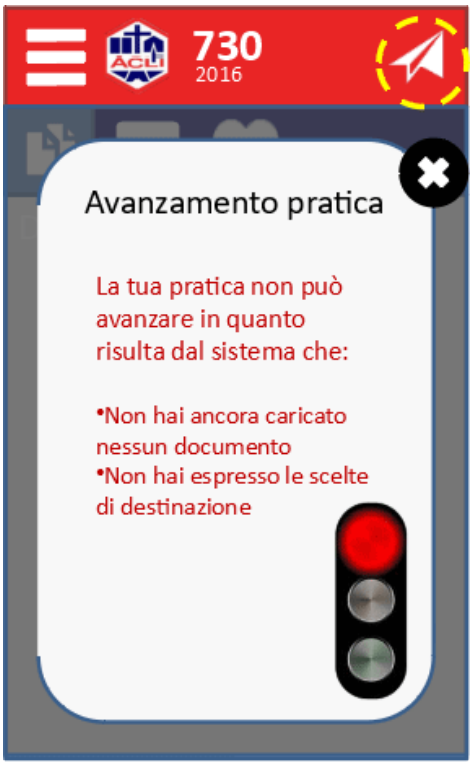

Figura 2.24: Status panel durante la fase di caricamento documenti.

La figura 2.24 mostra i consigli che vengono dati dal sistema all'utente durante la fase di input.

Per procedere alla elaborazione della sua pratica deve aver caricato dei documenti ed aver espresso tutte le scelte di destinazione.

In alcuni momenti dell'anno può capitare che il committente decida di aprire la campagna stagionale prima della delibera delle normative che regolamentano la materia. In quel caso viene mostrato all'utente un messaggio che lo informa che non potrà ancora trasmettere la sua pratica prima di una certa data.

Figura 2.25: Invio della richiesta di elaborazione della pratica.

Figura 2.26: Invio della richiesta di annullamento dell'elaborazione.

La figura 2.25 mostra il contenuto dello status panel che compare quando durante la fase di caricamento dei documenti la pratica ha ottenuto tutti i requisiti per poter essere inviata all'operatore per la sua elaborazione.

Una volta che il contribuente decide in *inviare* la sua pratica questa entra in una fase in cui si attende che l'operatore la prenda in carico.

Durante questo lasso di tempo non vi è alcuna elaborazione in corso, quindi di fatto il contribuente può continuare ad apportare modifiche. Qualora si accorgesse però di aver inviato la richiesta troppo presto e di avere bisogno di più tempo può arrestare il processo tramite lo status panel (vedi figura 2.26).

Se invece il contribuente non eseguirà l'arresto, l'operatore (che è stato allertato nel frattempo) deciderà di prendere in carico la pratica e questa entrerà in stato 'IN ELABORAZIONE' (vedi diagramma degli stati 2.8)

A questo punto il contribuente non potrà fare altro che aspettare l'elaborato. Se per qualche motivo decidesse ora di arrestare l'elaborazione dovrebbe farlo dialogando tramite la chat con l'operatore per giustificare l'intenzione di questa richiesta.

Approvazione Quando l'elaborazione è terminata al contribuente compare una nuova tab `inbox`. Attraverso di essa può accedere alla bozza dell'elaborato in modo da prenderne visione e decidere se approvarla.

L'atto dell'approvazione ha il senso di una firma e permette all'operatore del CAF di reputare buono l'elaborato definitivo.

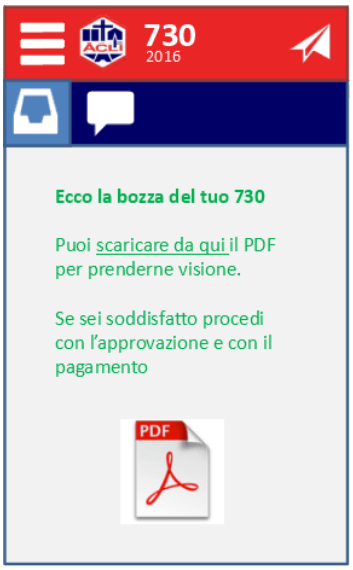

Figura 2.27: Tab inbox.

Figura 2.28: Approvazione dell'elaborato.

Una volta che il contribuente ha proceduto all'approvazione e al pagamento, nella sua inbox comparirà l'opzione per effettuare il download del documento definitivo che verrà trasmesso telematicamente agli organi fiscali.

2.3 Sviluppo

2.3.1 Implementazione del sistema 'app'

L'applicativo è stato sviluppato con una versione del framework AngularJS maggiore della `1.5`. Questo dettaglio sulla versione è importante in quanto da quella versione in avanti è stato introdotto il supporto ai `components`[15] che permettono di definire in maniera concisa delle entità che hanno una rappresentazione a video (view) ed un comportamento associato (controller).

Questo costrutto è molto espressivo; sostanzialmente tutto l'aspetto visuale e la gestione dell'interazione con l'utente sono state implementate in questo modo, organizzando l'applicativo come una gerarchia di componenti.

La libreria di routing precedentemente descritta permette di definire gli stati del grafo dell'applicazione e per ognuno di essi disporre quali componenti devono essere istanziati ed integrati nella vista. É in questo modo che è stata implementata la funzionalità che caratterizza le Single Page Applications.

L'implementazione della *domain logic* ed il modeling delle risorse trattate dal sistema è stata fatta definendo dei `services`[16].

Il linguaggio JavaScript ha una natura object oriented particolare e per questo motivo **AngularJS offre 3 differenti modi per definire un service**; questi modi sono detti anche *recipe*. Indipendentemente dal modo usato, a runtime un servizio avrà sempre la stessa natura: ossia quella di un oggetto. Ciò che viene differenziato è invece il come questa istanza è stata ottenuta: da qui il termine *ricetta*.

Le recipes che permettono di definire un servizio in AngularJS sono le seguenti:

Factory Il service consiste nell'oggetto ritornato da un metodo.

Durante la fase di definizione si associa il metodo factory all'identificativo del servizio. Quando a runtime una qualche entità dichiara questo servizio come una sua dipendenza, l'*injector* provvede ad eseguire il metodo factory per ottenere l'istanza del servizio ed iniettarla come dipendenza.

Service Il nome crea una certa ambiguità ma il significato è il seguente: con la recipe 'service' si può definire un service istanziabile attraverso una *contructor function*.

Durante la fase di definizione all'identificativo del servizio viene associata una funzione costruttore JS. Quando a runtime questo dovrà essere iniettato, tale funzione verrà invocata con l'operatore `new`.

Provider É un caso particolare di servizio che permette di definire una componente iniettabile a run-time ed una iniettabile a config-time.

[15] *AngularJS Components* https://docs.angularjs.org/guide/component
[16] *AngularJS Services* https://docs.angularjs.org/guide/services

Questa caratteristica permette di definire dei service che sono parametrizzabili prima di essere istanziati e serviti alle entità che dipendono da essi.

Ad esempio il servizio che si occupa di restringere le immagini uploadate è stato codificato sotto forma di provider in modo che durante la fase di configurazione si potesse esprimere la misura delle immagini da produrre.

Quello che seguirà ora è una rassegna dei problemi che si sono presentati più frequentemente durante lo sviluppo de IL 730 ONLINE, indicando per ognuno lo stile implementativo adottato ed alcuni esempi di pseudo codice.

2.3.1.1 Definizione dei *components* di AngularJS

I componenti di AngularJS permettono di definire nuovi tag estendendo il linguaggio di markup HTML. A questi tag è assegnata una vista che corrisponde al template indicato in fase di definizione del component. La vista può accedere tramite la variabile magica `$ctrl` al controller definito per quel componente e di conseguenza ai suoi metodi ed ai suoi *binding*. Questi sono dati passati dall'esterno al componente attraverso il protocollo di input stabilito (vedi nell'esempio che segue la parametrizzazione `@` per gli input 'stringa', e quella `<` per i reference in lettura).

```
1  (function(){
2    var theController = function(StringUtils){  //inject utils into ctrl
3      var ctrl = this;
4
5      ctrl.getCaseCorrectedSignature = function(){
6        return StringUtils.toTitleCase(ctrl.signature);
7      };
8    };
9
10   var theComponent = {
11     controller:   theController,
12     templateUrl:  'template.html',
13     bindings: {
14       hint:       "@",  //Text to be shown while not signed
15       signed:     "<",  //whether to show the signature or not
16       signature:  "@"   //es 'Maria Rossi'
17     }
18   };
19
20   angular.module('il730online.kit.ui.signatureStrip').component(
21       'signatureStrip',
22       theComponent);
23 })();
```

Listing 2.1: Definizione del componente <`signature-strip`> che mostra la firma del contribuente (visibile in opera nella videata scelte di destinazione 2.22).

Sul lato frontend AngularJS permette di *interpolare* i valori dinamici esposti dal controller per stamparli a video attraverso l'operatore di *doppia parentesi graffa*.

Il framework mette inoltre a disposizione un corredo di direttive che possono essere attivate per modificare il contenuto del documento semplicamente aggiungendo al markup opportuni attributi `ng-*`. Nell'esempio che segue sono stati usati:

- L'attributo `ng-class` per assegnare condizionalmente la classe 'signed' al tag div su cui è applicato.
- L'attributo `ng-show` per mostrare o nascondere condizionalmente il blocco su cui è applicato in funzione del binding con il controller di tipo boolean in input.
- L'attributo `ng-animate` per applicare la famiglia di stili css *fade* mentre è in corso l'apparizione o la scomparsa dell'elemento.

```
1  <div class="wrap" ng-class="{signed: $ctrl.signed}">
2
3    <div class="hint">
4      {{$ctrl.hint}}
5    </div>
6
7    <div     class="signature"
8             ng-show="$ctrl.signed"
9             ng-animate="'animate-fade'">
10
11     {{$ctrl.getCaseCorrectedSignature()}}
12   </div>
13
14 </div>
```

Listing 2.2: Template HTML del componente <`signature-strip`>.

Questo componente elementare è comunque sufficiente a dimostrare come con pochissimo codice sia in realtà possibile creare delle funzionalità complesse: in particolare questo esempio mostra la comparsa condizionale di elementi, la loro animazione grafica con un piacevole effetto fade e l'implementazione del protocollo di passaggio valori inter-componente.

Questa elementarità permette di scomporre l'applicazione in tantissimi componenti di più facile gestione e riuso (ad esempio questo componente viene impiegato sia nella vista della scelta dell 8, del 5 e del 2 per mille) in maniera per lui completamente indipendente al contesto.

Il foglio di stile non verrà mostrato per motivi di spazio ma il fatto di avere un tag specifico (in questo caso <`signature-strip`>) permette di scrivere regole molto focalizzate, quindi è difficile che possano crearsi side effects stilistici.

2.3.1.2 Definizione degli *states* di UI-Router

L'implementazione del meccanismo di navigazione della SPA è ottenuto attraverso l'uso del router open source `UI-Router`. Questo non fà parte del core di AngularJS quindi gli *stati* dell'applicazione devono essere dichiarati attraverso il *provider* messo a disposizione da questo modulo, iniettandolo durante la fase di *config*.

Il fatto di eseguire la registrazione di tutti gli stati in questa fase permetterà al router di avere un quadro d'insieme completo del grafo di navigazione quando l'applicazione entrerà in fase di *run*.

Per questi motivi il processo di dichiarazione di uno stato viene codificato all'interno di un *handler* registrato in modo tale che il framework vada ad attivarlo prima di iniziare l'esecuzione dei run-blocks[17].

JavaScript è un linguaggio non tipato, quindi la struttura di un oggetto che dichiara uno stato deve *ricalcare* quella prevista dalla documentazione ufficiale[18].

Alcune proprietà che possono essere definite per una *State Declaration* sono:

name Consiste in una stringa composta da parole separate da punti. L'ultima parola della sequenza è il *nome* dello stato mentre la cascata di nomi precedenti identifica la *posizione* in cui deve essere collocato nel grafo.

es. `app.userDomain.workspace.modello730.single`

url Identifica la *porzione di URL* che deve essere mappata su questo stato. Quando si approda dall'esterno all'applicazione, il router intercetta la richiesta ed interviene attuando una sequenza di transizioni. Gli stati toccati da questa 'attraversata' sono quelli le cui proprietà `url` ricongiunte corrispondono alla richiesta dall'utente.

Questa proprietà può essere definita come una stringa fissa da matchare oppure come un segnaposto da usare per catturare un valore parametrico (es. `:year`)

resolve Permette di definire quali oggetti devono essere recuperati nel momento in cui si entra in questo stato. I dati *risolti* possono essere iniettati nei componenti sotto forma di binding.

onEnter/onExit/redirect Lo stato può definire delle logiche *di azione* da eseguire quando si entra in esso o lo si ha abbandonato; oppure logiche *di filtro* come ad esempio quelle per evitare l'ingresso nello stato eseguendo un redirect.

view L'applicazione ha un template di partenza che è quello del file `index.html`. Dentro questo file deve essere presente obbligatoriamente un'occorrenza della direttiva `<ui-view>`. Questo tag definisce l'*outlet* in cui il router caricherà le viste durante

[17] *Order of execution* https://docs.angularjs.org/guide/module#dependencies-and-order-of-execution
[18] *State Declaration Interface* https://ui-router.github.io/ng1/docs/latest/interfaces/state.statedeclaration.html

la navigazione. Ogni componente può a sua volta definire nuovi outlet all'interno del suo rispettivo template: in questo modo per ogni stato dell'applicazione sono disponibili on screen una serie di outlets per i quali si può specificare quale componente installarvi.

Uno stato non deve necessariamente specificare delle views: in quel caso quanto percepito dall'utente a video sarà il risultato della composizione dei templates attivati dagli stati parent.

```
(function(){
  var onConfig = function($stateProvider){

    var stateDeclaration = {
      name:      'app.userDomain.workspace.modello730.single',
      url:       '/:year',   //es. '2017'

      onEnter:  function(modello730, Modello730Stato, $state){
        if(modello730.idStato == Modello730Stato.ANNULLATA){
          //Accesso Negato: redirect all'archivio
          $state.go("app.userDomain.workspace.modello730.archive");
        }else if(modello730.idStato == Modello730Stato.CREATA){
          //Apertura della pratica da completare: redirect al wizard
          $state.go(
            "app.userDomain.workspace.modello730.single.
               completeRegistration");
        }
      },

      resolve: {
        modello730: function(modello730List, $transition$){
          //cerca nel model in base all'anno richiesto via url
          return modello730List.searchByYear(
              $transition$.params().year);
        }
      }
    };

    $stateProvider.state(stateDeclaration);
  };

  angular
    .module('il730online.app.userDomain.workspace.modello730.single')
      .config(onConfig);
})();
```

Listing 2.3: Esempio di dichiarazione di uno stato in cui si evidenzia la possibilità di *risolvere* un oggetto partendo da una porzione dell'URL e di eseguire delle logiche di verifica su di esso.

2.3.1.3 Implementazione dei singletons con il recipe *service* di AngularJS

Spesso è risultato necessario garantire che di certe entità esistesse una sola istanza in circolazione e che questa fosse facilmente accessibile da qualunque punto del progetto.

Sono un esempio di queste casistiche le **utilities**.

La soluzione più naïve sarebbe quella di implementare delle procedure statiche in classi accessibili globalmente. Questa staticità impedirebbe però di fare estensioni di queste classi e realizzare delle specializzazioni di esse.

Adottando invece il pattern *singleton* [3] non si precluderebbe questa possibilità ma si introdurrebbe una dipendenza *hardcoded* sulla concretizzazione, rendendo il codice sviluppato meno testabile.

La soluzione ottima consiste nello sfruttare il meccanismo di DI di AngularJS definendo dei *servizi* attraverso la recipe `service`[19].

```
1  (function(){
2    var constructorFunction = function (/* no arguments */){
3
4      var me = this;
5
6      /**
7       * Capitalizes the first letter of each word
8       *
9       * @param {String}    str     es. joHn sMitH
10      * @returns {String}  es. John Smith
11      */
12     me.toTitleCase = function(str){
13
14       return str.replace(
15           /\w\S*/g,
16           function(txt){
17             return txt.charAt(0).toUpperCase() + txt.substr(1).
                   toLowerCase();
18           });
19     };
20
21     [...]
22   };
23
24   angular.module('commons.kit').service(
25       'StringUtils',
26       constructorFunction);
27 })();
```

Listing 2.4: Esempio di implementazione di una classe utility come service singleton.

[19] *AngularJS Service Recipe* http://www.learn-angular.org/#!/lessons/the-service-recipe

2.3.1.4 Simulazione delle classi JS con il recipe *factory* di AngularJS

JavaScript è un linguaggio object-oriented e nonostante supporti un tipo di dato chiamato oggetto, non ha una nozione formale di classe [14].

I linguaggio offre strumenti sufficienti per colmare questa mancanza, riuscendo a simulare le features dei linguaggi class-based come Java e C++.

Tipicamente in questi linguaggi più evoluti la *classe* definisce la struttura degli oggetti di quel tipo specificando quali sono le proprietà, i rispettivi tipi di dato ed i metodi che espone.

JavaScript non ha dei costrutti per definire tali tipi di dato, ma permette di approssimarne questo comportamento con i *costruttori* ed i loro oggetti *prototype*.

L'idea è quella di definire un oggetto che ha il senso di classe e di usarlo come *master* per *stampare* dei suoi simili. Chiaramente questo oggetto deve essere conservato inalterato per tutto il ciclo di vita dell'applicazione, altrimenti da un certo momento in avanti le nuove istanze inizierebbero ad avere sembianze diverse da quelle generate prima di manomettere l'oggetto-classe.

Questo è uno dei motivi per cui per fare un uso avanzato del linguaggio JS serve disciplina.

Durante lo sviluppo de IL 730 ONLINE sono emersi dei problemi risolvibili con un approccio classico basato su catene ereditarie di classi.

L'esempio che viene riportato è quello della gestione dei task di upload.

Il sistema permette di effettuare invii di documenti in diverse fasi della procedura: durante l'iscrizione (dove il documento inviato risulta collegato al contribuente) e durante la gestione della pratica (dove questo risulta collegato al modello 730).

La funzionalità di fondo è la stessa: per entrambi gli scenari bisogna pre processare il file in ingresso, elaborare un checksum, spedire e gestire gli eventuali errori, dando al contempo un feedback e gestendo l'eventuale richiesta di annullamento dell'upload.

Ciò che differenzia un tipo di task dall'altro è il corredo di informazioni da veicolare insieme al *payload*, necessario lato server per abbinarlo alla giusta entità.

Per questo scopo si è messo in piedi un meccanismo di eredità *prototype-based* tipico del linguaggio JavaScript, in cui l'oggetto master (ossia *la classe*) è definito come un service di AngularJS tramite il *recipe factory*.

Lo stile di definizione di classi basato sui prototype è quello più flessibile perché permette di supportare (come vedremo nei capitoli successivi) features avanzate ma il prezzo da pagare è quello di perdere l'incapsulamento. Tutte le proprietà interne dell'oggetto devono essere definite nel prototype: pena la perdita delle stesse nel momento in cui verrà fatto il subclassing.

La convenzione è quella di *etichettare* le proprietà interne prefissando il loro nome con un underscore. Sta nell'utilizzatore del codice avere la disciplina di onorare questa specifica evitando di mettervi mano.

```
(function(){
  var theFactory = function(imageResizer, fileMd5Service, ...){

    //Costruttore: inizializza le variabili interne
    var AbstractUploadTask = function(){
      this._state = [...];
    };

    /*** INTERNALS ***/

    //Esempio di metodo interno (notare l'underscore)
    AbstractUploadTask.prototype._maybeResizeImage = function(){
      [...]
    };


    /*** ABSTRACT INTERFACE ***/

    //Esempio di metodo astratto
    AbstractUploadTask.prototype._getUploadServiceData = function(){
      throw "this must be implemented by concrete class";
    };

    /*** PUBLIC INTERFACE ***/

    AbstractUploadTask.prototype.getState = function(){
      return this._state;
    };

    AbstractUploadTask.prototype.start = function(){
      [...]
    };

    return AbstractUploadTask;
  };

  angular.module('il730online.core.uploads').factory(
      'AbstractUploadTask',
      theFactory);

})();
```

Listing 2.5: Esempio di implementazione di una classe abstract come service factory.

2.3.1.5 Polimorfismo con JavaScript ed AngularJS

In questo capitolo verrà descritto come sono state colmate delle mancanze del linguaggio JavaScript per ottenere funzionalità analoghe a quelle dei linguaggi object oriented più evoluti.

Il linguaggio JS adotta una gestione dei tipi di dato detta *Duck Typing*[20] che consiste sostanzialmente nel definire implicitamente un concetto di *tipo* in base alle funzioni che l'oggetto è in grado di supportare. Ad esempio se l'oggetto *aeroplano* e l'oggetto *piccione* hanno entrambi un metodo *vola*, un ipotetico codice che opera su oggetti di tipo *Volante* potrà operare su entrambi interscambiabilmente.

Una delle particolarità di questo linguaggio sta proprio nel fatto di riuscire a supportare il *polimorfismo* senza che l'informazione su quali sono *i tipi* coinvolti sia definita in alcun luogo.

Questa forte caratteristica di ***weak typedness*** **implica che in JavaScript il contratto d'interfaccia dei vari tipi risieda solo nella mente dello sviluppatore**. Come già detto più volte in questa tesi, questo linguaggio richiede molta disciplina da parte del suo utilizzatore, perché il corretto funzionamento del codice dipende dalla sua capacità di rispettare queste relazioni invisibili.

Gli oggetti JavaScript non solo non hanno una nozione di tipo ma possono anche cambiare la loro forma a runtime in quanto **il linguaggio permette di decorare un oggetto aggiungendo o sostituendo proprietà e metodi durante l'esecuzione**.

Questa feature unita allo stile di definizione di *classi* basata sul prototype mostrata nel capitolo precedente, permette di definire gerarchie di classi.

Ora verrà descritto l'iter attraverso il quale è stato possibile **definire nel contesto dell'applicativo AngularJS una famiglia di classi le cui istanze potevano essere trattate come se fossero state di uno stesso tipo di base**.

1. Si individua il tipo di dato comune su cui deve operare la procedura e si definisce concettualmente la sua interfaccia.

   Un'esempio ne IL 730 ONLINE sono stati i task di upload. Diverse parti del programma dovevano gestire degli upload e con significati diversi ma ognuno di questi ambiti era accomunato dalle stesse operazioni consentite all'utente (selezione del file, avvio, annullamento) e gli stessi feedback (progresso dell'upload, trasmissione interrotta, file ricevuto corrotto).

   Il tipo di dato scelto per presentare un esempio è dunque il *task di upload*.

2. Si definisce un oggetto che ha il senso della classe base e lo si assegna con una funzione costruttore: in questo modo si può usare l'operatore `new` su di esso.

[20] *Duck Typing* https://en.wikipedia.org/wiki/Duck_typing

L'oggetto in questione è `AbstractUploadTask` e deve essere definito attraverso un service AngularJS in modo che questa *classe* possa essere resa disponibile globalmente attraverso il meccanismo della Dependency Injection.

Di fatto la mancanza del linguaggio nel definire staticamente una classe viene colmata definendo un oggetto singleton che la modella e condividendolo a livello dell'applicazione per usarlo come *stampo* di istanze.

3. La recipe da usare per la definizione di questo service deve essere quella factory. In questo modo è possibile eseguire del codice prima di *pubblicare* il service.

   Il codice in questione ha lo scopo di inizializzare il prototype di questo oggetto master in modo che disponga di tutti i metodi previsti dall'interfaccia.

4. Si definiscono allo stesso modo altri services Angular che modellano le sottoclassi (es. `ContribuenteAllegatoUploadTask`, `Modello730UploadTask` eccetera)

5. Ognuno di questi dichiara come dipendenza il service definito al punto 2 così da disporre dell'oggetto 'classe base' durante la definizione degli oggetti 'sotto-classe'.

   Lo scopo di tutto questo è qello di inizializzare i prototype degli oggetti 'sottoclasse' con quello della classe base:
   `Modello730UploadTask.prototype = AbstractUploadTask.prototype`

   Il costruttore della classe derivata applica inoltre la funzione costruttrice della classe base sulla propria istanza, così da inizializzare le proprietà appena 'ereditate' tramite copia del prototype.

6. A questo punto ogni sottoclasse decora il suo prototype precedentemente inizializzato con le sue specializzazioni.

   Si possono aggiungere nuovi metodi per supportare un uso diretto della sottoclasse o sostituire dei metodi già esistenti per supportare un uso polimorfico attraverso l'interfaccia di base.

```
(function(){
  var theFactory = function(AbstractUploadTask){  //inject base class

    //Sub class constructor
    var Modello730AllegatoUploadTask = function(
              idModello730, classeAllegato, statoAllegato){

      //Invoke base class constructor
      AbstractUploadTask.call(this);

      //Remember parametrization
      this._idModello730 = idModello730;
      this._classeAllegato = classeAllegato;
      this._statoAllegato = statoAllegato;
    };

    //Inherit from base class
    Modello730AllegatoUploadTask.prototype =
              AbstractUploadTask.prototype;

    /*** ABSTRACT INTERFACE IMPL ***/

    //Replace inherited (stub) implementation with concrete one
    Modello730AllegatoUploadTask.prototype.
    _getUploadServiceData = function(){

      //using sub-class properties
      return {
          idModello730:                 this._id,
          idModello730AllegatoClasse:   this._idAllegatoClasse,
          idModello730AllegatoStato:    this._idAllegatoStato};
    };

    return Modello730AllegatoUploadTask;  //return sub class
  };

  angular.module('il730online.core.uploads').factory(
      'Modello730AllegatoUploadTask',
      theFactory);
})();
```

Listing 2.6: Esempio di definizione di una sottoclasse che specializza una classe base abstract usando il meccanismo dell'ereditarietà prototype based, supportata dalla Dependency Injection di AngularJS.

2.3.1.6 Implementazione delle *resources* ed override di metodi statici con JavaScript ed AngularJS

Nei capitoli precedenti si è parlato di come sia da evitare l'uso di metodi statici preferendo al loro posto metodi di istanza di un oggetto singleton *utility* condiviso globalmente.

La motivazione di fondo è sostanzialmente quella di non precludere l'uso delle tecniche polimorfiche appena discusse, che abilitano la possibilità di sostituire l'implementazione (mocking) e di fare specializzazioni del servizio.

Nonostante questa filosofia sia quella da perseguire in generale, và fatto notare che **in certi scenari di programmazione AngularJS risulta molto utile poter fare l'override di metodi statici**: uno di questi è l'implementazione di client REST basati sui servizi offerti dal modulo `ngResource`[21].

Questo modulo (sviluppato dal core team di AngularJS) attraverso il servizio `$resource` permette di definire delle **classi che modellano le risorse remote esposte da un'API REST** oscurando all'utilizzatore i dettagli delle richieste AJAX e del formato degli URL [11].

Le istanze di queste classi modellano una risorsa specifica ed offrono dei metodi canonici per agire su di essa (delete, update eccetera). Questo funzionamento è molto simile a quello descritto dal pattern *Active Record* (1.1.1.4).

Le operazioni di interrogazione (cioè che ritornano una lista di istanze) non agiscono su una risorsa specifica quindi non sono disponibili come metodi d'istanza ma bensì come metodi statici della classe.

```
1  //Define a CreditCard class that models a remote resource
2  var CreditCard = $resource(...);
3
4  //Creation operation  (instance method)
5  var cc = new CreditCard();
6  cc.number = "1234xxxxxxxxx";
7  cc.name = "Some User"
8  cc.$save(); //POST: /creditCards {number: '1234xx', name: 'Some User'}
9
10 //Query operation (static method)
11 var allCCs = CreditCard.query();  //GET: /creditCards
```

Listing 2.7: Esempio di impiego del service `$resource` per generare una classe che modella una risorsa remota accessibile via API REST e di come usare tale classe.

In diversi scenari **è risultato necessario *decorare* i risultati di queste interrogazioni**: per fare questo senza introdurre un'API aggiuntiva occorreva fare l'override del metodo statico ma conservando anche l'implementazione originale (che è opaca).

[21] *Modulo ngResource di AngularJS* https://docs.angularjs.org/api/ngResource

Questo obiettivo è stato raggiunto escogitando una particolare soluzione basata sulle *funzioni anonime immediatamente eseguite* dette *IIFE*[22] di JavaScript.

1. Il metodo statico viene rimpiazzato (overridden) con il risultato dell'esecuzione di una IIFE.
2. Alla IIFE viene passato come parametro il metodo statico stesso. Al momento in cui questa funzione anonima verrà eseguita, il reference ricevuto come argomento corrisponderà all'implementazione originale (che chiameremo *superGet*).
3. Nel body della IIFE viene definito il nuovo metodo che andrà a sostituire quello originale: lo chiameremo *newGet*. Questo viene dato come valore di ritorno dell'esecuzione così che possa propagarsi come descritto al primo punto.
4. L'implementazione di *newGet* può invocare l'implementazione di base, in quanto il reference a *superGet* è rimasto intrappolato nella sua *chiusura*[23] e continuerà ad essere disponbile lì dentro anche dopo che è avvenuto con successo l'assegnamento all'esterno ed è andato a tutti gli effetti perduto l'ultimo reference pubblico all'implementazione originale.

```
1  Modello730.get = (function(superGet){
2
3    var newGet = function(a, b, c, d){ //expecting up to 4 args
4
5      //invoking base class implementation
6      var superResult = superGet.call(this, a, b, c, d);
7
8      //performing our customization on results
9      var newResult = decorate(superResult);
10
11     //returning decorated results
12     return newResult;
13   };
14
15   return newGet;  //returning new impl
16
17 })(Modello730.get); //passing original implementation
```

Listing 2.8: Esempio avanzato di override dei metodi statici presenti nelle classi generate con il service `$resource`.

Il metodo statico nell'esempio mantiene l'interfaccia standard delle classi 'resource', ritornando però un dato *custom*, senza aver dovuto riscrivere l'implementazione di base. es. `var customResult = Modello730.get();`

[22] *Immediately Invoked Function Expr* https://developer.mozilla.org/en-US/docs/Glossary/IIFE
[23] *Chiusure JavaScript* https://developer.mozilla.org/it/docs/Web/JavaScript/Chiusure

2.3.2 Implementazione del sistema 'platform'

Prima di discutere le scelte implementative fatte per la componente lato server del sistema và fatta una precisazione di carattere storico.

La versione del framework Yii 1.1.x è quella usata per il progetto. Sebbene sia attivamente mantenuta, il suo design originale risale intorno all'anno 2010.

A quel tempo le funzionalità object oriented del linguaggio PHP erano state introdotte da pochissimo tempo: ad esempio la funzionalità dei *namespaces* è presente a partire dalla versione 5.3 di PHP rilasciata nell'anno 2009.

Probabilmente il team di design all'epoca ha scelto di abbracciare le novità con moderazione in modo da consentire l'uso del framework anche in quegli hosting meno aggiornati.

Questo fatto ha delle implicazioni nel modo in cui deve essere costruita l'applicazione e nel modo in cui devono essere strutturati i sorgenti.

2.3.2.1 Organizzazione del file system ed *Inversion Of Control*

Il framework Yii permette di sviluppare diversi tipi di applicazioni con il linguaggio PHP. Ad esempio oltre al tradizionale tipo di applicazione web a *roundtrip* (1.3.2.1) permette di scrivere delle *console app* che hanno come concetto di input gli argomenti di invocazione a linea di comando e come output quello testuale della shell.

Questa carattersitica di isolamento tra logica interna di elaborazione e frontiera di input è molto interessante perché permette di **codificare delle azioni che sono agnostiche al contesto che ne ha scaturito l'attivazione**.

Questa impostazione dell'applicazione segue il pattern *Inversion Of Control*[24] in quanto il codice applicativo non guida il flusso del sistema ma viene passivamente attivato quando opportuno.

Di fatto la prima tematica da curare quando si inizia lo sviluppo di una applicazione Yii è quella di **organizzare i propri elementi del progetto in modo che questi siano attivabili dal framework**.

Come anticipato nella premessa di questo capitolo, l'uso dei namespaces non è al core del funzionamento quindi l'identificazione di un componente non è basata sulla sua organizzazione logica ma bensì su quella a livello del file system: affinché una richiesta venga correttamente smistata occorre collocare i sorgenti con una **opportuna organizzazione a livello di cartelle**.

Il framework Yii favorisce le convenzioni al posto delle configurazioni[25] e presuppone che la directory dell'applicazione contenga una serie di cartelle di default usate per vari scopi.

[24] *Pattern Inversion of Control* https://it.wikipedia.org/wiki/Inversione_del_controllo

[25] *Yii Conventions* https://www.yiiframework.com/doc/guide/1.1/it/basics.convention#cartelle

Questo template organizzativo può essere ripetuto in maniera nidificata così da suddividere l'applicazione principale in sotto unità autosufficienti, ognuna delle quali può essere dotata di models, controllers, viste ed altri componenti di supporto.

Le unità appena descritte sono chiamate *moduli* e sono state impiegate per differenziare il comportamento del sistema in base ai client, condividendo però un'unità elaborativa centralizzata.

Ricordiamo che questa *necessità di differenziazione* è derivata dalla volontà di non precludere un giorno la portabilità del frontend come app mobile nativa. In quell'ambito i cookie non sono supportati e la gestione della sessione deve essere realizzata in una maniera alternativa. I requisiti implementativi però imponevano di realizzare il backoffice di amministrazione con il tradizionale stile di web application roundtrip per il quale lo strumento dei cookie è irrinunciabile.

Questa differente gestione delle sessioni, unita alla necessità di produrre output per le richieste in formato differente hanno spinto ad implementare la parte `dashboard` e quella `API` sotto forma di moduli Yii separati.

Quando la richiesta interessa un controller organizzato all'interno di un modulo, il *flusso di attivazione* operato dal framework passa prima per il modulo stesso. É in quell'occasione che è possibile eseguire quelle logiche e quelle configurazioni comuni per quell'intera famiglia di controllers.

2.3.2.2 Implementazione della *Domain Logic*

Il sistema si occupa sostanzialmente di gestire un'anagrafica di contribuenti ed un archivio di files. Le logiche connesse a questo ambito sono quelle tipiche del **trattamento dei dati**, ossia la creazione ed il recupero di unità informative persistenti, l'attuazione di logiche di validazione sui dati e la verifica del rispetto di certe regole di organizzazione degli stessi.

Un'altra sfera della gestione è invece rappresentata dalla **comunicazione**. Questo aspetto è più orientato al processo e solo indirettamente ha a che fare con dati persistenti. Stiamo parlando in particolare dell'invio di comunicazioni chat, delle loro relative notifiche, e dell'invio di sms ed email.

Ora verranno passati in rassegna gli elementi principali del sistema descrivendo come questi sono stati implementati nel contesto di un'applicazione Yii.

models Le informazioni relative alle varie entità del sistema (contribuenti, pratiche, allegati eccetera) sono memorizzate in tabelle di un database relazionale MySQL[26].

Attraverso il tool `Gii` (1.4.2.1) è stato possibile generare in automatico delle classi per modellare queste entità. Le classi generate derivano da `CActiveRecord`[27] che è la classe base del framework per rappresentare dati relazionati e si occupa di tutte le operazioni CRUD su di essi. Questa classe estende a sua volta `CModel` che è un'altra classe base del framework che rappresenta dati che richiedono validazione.

named scopes parametrici Un *named scope*[28] rappresenta un criterio che può essere concatenato con altri ed applicato nel contesto di una interrogazione per eseguire regole di filtraggio e di relazionamento avanzate senza scrivere query SQL.

Quando serve fare delle estrazioni di dati (select) piuttosto che implementare delle query specifiche (che risulterebbero complesse da scrivere ed assolutamente non riusabili) conviene implementare i singoli criteri come unità indipendenti e poi comporli per risolvere ogni specifica necessità.

Nella sostanza si definiscono sul model dei metodi che hanno un nome descrittivo del criterio che implementano usando verbi al tempo participio (come ad esempio `appartenenteA(owner)`, `inStatoAnnullato()`, `creatoIl(d)` eccetera); ognuno di questi metodi deve modificare l'oggetto di tipo `CDbCriteria` aggiungendovi le clausole che li caratterizzano e ritornando l'istanza del model, così che sia possibile concatenare queste chiamate con una sintassi *fluente*[29].

Alla fine si invoca uno dei metodi `findXXX` per eseguire la query consumando l'oggetto *criteria* compilato fino a quel momento e ritornando il *resultset*.

[26] *MySQL* https://www.mysql.com/it/
[27] *CActiveRecord* https://www.yiiframework.com/doc/api/1.1/CActiveRecord
[28] *Named Scopes* https://www.yiiframework.com/doc/guide/1.1/en/database.ar#named-scopes
[29] *Fluent Interface* https://en.wikipedia.org/wiki/Fluent_interface

```
1  class Modello730Allegato extends CActiveRecord{
2    [...]
3
4    public function inStato($idStato) {
5      $this->getDbCriteria()->mergeWith(array(
6        'condition' => 'idStato = :idStato',
7        'params' => array(':idStato' => $idStato)
8      ));
9
10     return $this;
11   }
12 }
```

Listing 2.9: Esempio di definizione di un named scope parametrico per filtrare gli allegati in uno specifico stato.

```
1  //Gli ultimi 3 allegati validi di Tizio
2  $allegati = Modello730Allegato::model()
3    ->collegatiAllaPratica($contribuenteTizio->modello730)
4    ->inStato(Modello730AllegatoStato::VALIDO)
5    ->limitatoA(3)
6    ->ordinatoPerDataDiUpload('ASC')
7    ->findAll();
8
9  /*
10   Equivalente alla query SQL:
11
12   SELECT
13     a.*
14   FROM
15     modello730 as m,
16     modello730_allegato as a,
17     contribuente as c
18   WHERE
19           m.idContribuente = c.id
20     and   a.idModello730 = m.id
21     and   c.id = 'contribuenteTizioId'
22     and   a.idStato = 1
23   ORDER BY a.timestampCrea ASC
24   LIMIT 3
25
26 */
```

Listing 2.10: Uso di named scopes parametrici per interrogare il database.

behaviors Tutti i *components* Yii (tra cui i models) supportano il pattern mixin[30] e permettono la possibilità di 'ereditare' dei metodi da componenti collegati *collezionando funzionalità piuttosto che specializzazioni.* La classe del framework `CActiveRecordBehavior`[31] è appositamente progettata per implementare logiche di contorno al trattamento dei dati senza operare fisicamente sulla classe del model.

In questo modo il model ha la stessa espressività che avrebbe avuto implementando direttamente al suo interno le varie logiche ma con il vantaggio che queste sono chiaramente esplicitate e logicamente organizzate in unità separate.

```
1  class Modello730StatoDocumentaleChangeGuardBehavior
2        extends CActiveRecordBehavior{
3
4    public function beforeSave($event){
5
6      $prop = $this->getOwner()->getModified('idStatoDocumentale')
7      $oldValue = $prop['old'];
8      $newValue = $prop['new'];
9
10     switch($oldValue){
11
12       //CARICAMENTO_DATI->DA_ELABORARE unica consentita
13       case Modello730StatoDocumentale::CARICAMENTO_DATI:
14
15         switch($newValue){
16
17           case Modello730StatoDocumentale::DA_ELABORARE: //ok
18             break;
19
20           default:  //tutte le altre proibite
21             $this->getOwner()->addError(
22                 'idStatoDocumentale',
23                 'Transizione non consentita');
24             break;
25         }
26
27         break;
28
29       //DA_ELABORARE->CARICAMENTO_DATI consentita
30       //DA_ELABORARE->IN_ELABORAZIONE consentita
31       case Modello730StatoDocumentale::DA_ELABORARE){
32
33         [...]
```

Listing 2.11: Stralcio del behavior che veglia sul rispetto delle transizioni ammissibili descritte nel diagramma 2.8.

[30] *Pattern Mixin* https://en.wikipedia.org/wiki/Mixin
[31] *CActiveRecordBehavior* https://www.yiiframework.com/doc/api/1.1/CActiveRecordBehavior

2.3.2.3 Implementazione dei servizi di comunicazione

SMS Service Il provider selezionato era Mobyt. Tra le varie API a disposizione è stata scelta quella di tipo Web Service SOAP per la facilità di generare un client partendo dal suo WSDL[32]. Il codice generato è stato *wrappato* sottoforma di componente Yii in modo da poterlo mockare durante le fasi di sviluppo e testing automatico.

Mailer Service É già stato giustificato il fatto di aver scelto l'uso del protocollo SMTP per disaccoppiare il servizio di spedizione dall'applicazione (e poter quindi testarla appoggiandosi ad un server fake). La libreria standard *de-facto* per l'invio di mail è PHPMailer e fortunatamente qualcuno ha già provveduto a distribuirla sotto forma di *extension*[33] per il framework Yii.

Dal momento che non è servito fare alcun wrapping, l'unica attività svolta è stata la predisposizione di due set di configurazioni: una per l'uso di produzione (parametrizzata per usare il server vero) ed una per lo scopo di debug/testing (parametrizzata per usare il server Mailcatcher presentato al capitolo 1.5.2).

Chat Il concetto alla base è la *stanza* ed ogni contribuente ne ha una. Questa ha dei *partecipanti* che sono lui (ovviamente) ed i membri dello staff del CAF. I *messaggi* hanno una nozione di *mittente* e sono in *ordine cronologico*.

L'implementazione è piuttosto banale dal punto di vista fisico: si tratta sostanzialmente di gestire una tabella del database in lettura e scrittura.

Ciò che invece non è banale é realizzare una meccanica di recapito realtime con basso consumo di risorse. Un'implementazione di questo tipo avrebbe richiesto un notevole sforzo costruttivo, in quanto il codice PHP viene eseguito nel mezzo di una richiesta-risposta e non si presta particolarmente per implementare soluzioni reattive ad eventi.

La scelta è ricaduta su un tradizionale meccanismo basato su polling, consci del fatto che la soluzione non era affatto ottimale data l'ipotesi di operare su grande scala. L'implementazione è stata ottimizzata il più possibile sul lato client accelerando e rallentando la frequenza di refresh in base al trend di ricezione degli ultimi messaggi. In questo modo quando l'utente sta attivamente chattando la consegna è reattiva, mentre quando non lo sta facendo il consumo di risorse sul server è estremamente ridotto.

Notifiche Un'utente del sistema (Contribuente o membro dello staff) può essere notificato per diverse ragioni (presenza di nuovi messaggi nella chat, presenza di nuovi documenti da visionare, cambi di stato della pratica eccetera). Tutti questi *eventi* sono collezionati in dei *canali* ed ogni utente può averne più di uno a lui intestati.

[32] *Mobyt SMS Provider WSDL* http://api.mobyt.it/mobws?wsdl
[33] *Yii Mailer Extension* https://www.yiiframework.com/extension/yiimailer

Quando si vogliono ricevere notifiche rispetto agli eventi che accadono in un certo canale si deve effettuare una *sottoscrizione*, diventando quindi un *sottoscrittore* di quel canale. Il servizio mantiene per ogni sottoscrizione un *puntatore* all'ultimo evento notificato in modo che sia possibile elaborare per ogni sottoscrittore la differenza tra l'ultimo evento accaduto e l'ultimo evento notificato. Questa differenza consiste proprio nell'elenco di eventi da notificare per quel canale di notifica.

C'è un momento in cui il sistema dovrà informare il sottoscrittore della presenza di questi eventi, quindi è stata sviluppata un'interfaccia (ad uso interno) per interagire con la meccanica delle notifiche ad un livello più alto di quello del database.

Il metodo `peek` serve per interrogare quali sono i nuovi eventi *senza consumarli*. Questa funzionalità viene impiegata ad esempio per mostrare il tipico alert numerico per informare l'utente della presenza di eventi non considerati. Quando avrà preso atto di queste novità, i rispettivi eventi non dovranno più comparire all'interrogazione successiva: questa operazione di 'lettura con consumo' è rappresentata dal metodo `take`.

Nel caso in cui certi eventi rimangano non consumati per un lungo periodo di tempo abbiamo previsto di inviare una mail automatica di riepilogo.

Questa feature aggiunge un livello ulteriore di sofisticazione in quanto le notifiche non saranno consumate dall'utente fino al suo successivo approdo al sistema ma il *daemon di reportistica via email* dovrà annotare di aver fatto questa segnalazione per non ripetere la cosa in continuazione.

Questa *annotazione* è sostanzialmente un ulteriore puntatore sul canale: quindi il daemon che opera per conto di un sottoscrittore, relativamente ad un canale, è un sottoscrittore del canale stesso!

L'implementazione fatta è decisamente complessa e sarebbe *off-topic* presentarla in questo contesto: ciò che è doveroso dire è che è stato lo sviluppo lato server più complesso ed è l'unico che ha *veramente imposto* la scrittura di Unit Tests.

2.3.2.4 Implementazione del modulo API

Il compito di questo modulo era quello di mettere a disposizione dei client una interfaccia di accesso e manipolazione alle entità del sistema.

L'implementazione si è liberamente ispirata allo stile REST ed è caratterizzata da questi aspetti principali:

- Ogni risorsa (o collezione di esse) è identificata da un URL *auto descrittivo*.
  Dal momento che la struttura degli URL ha delle implicazioni per quanto riguarda il routing delle actions operato dal framework Yii, è risultato necessario ideare una struttura di nomi *flat*.

- La manipolazione delle risorse avviene attraverso *rappresentazioni*. Per questo scopo è stato usato massivamente il formato `JSON`, se non per quei casi in cui erano coinvolti dati binari (es. pdf).

- Il metodo `POST` (invocato sull'URL di una collezione) crea una nuova risorsa. Quelli `PUT` e `GET` operano su URL che rappresentano una risorsa specifica e rispettivamente hanno lo scopo di modificarla o di ritornarne una rappresentazione.

- I metodi `GET` e `PUT` sono idempotenti, nel senso che una loro ripetuta invocazione non produrrà effetti diversi da quelli dati da una invocazione singola.

Alcuni esempi:

> `GET https://platform.il730.online/api/v1/modello730/`
> Ritorna le pratiche del contribuente loggato

> `GET https://platform.il730.online/api/v1/modello730/45`
> Ritorna la pratica con id=45 (verificando che sia del contribuente loggato)

> `POST /api/v1/Modello730Allegato/`
> Crea una risorsa 'allegato' acquisendo i dati ricevuti ed organizzandola come specificato dal body della richiesta:
> `{idModello730, idModello730AllegatoClasse, idModello730AllegatoStato, data}`

Scegliere questo stile è risultato utile perchè per via della sua 'standardizzazione' ha permesso l'implementazione del client con l'uso di librerie esterne.

Và detto però che in alcuni casi non è stato possibile implementare il servizio richiesto aderendo ai dettami di questo stile architetturale.

Siccome i browser non effettuano i download di file partendo da richieste AJAX, si è dovuto ricorrere in quei casi ad una meccanica di richiesta tradizionale basata su action&form, emettendo quindi richieste POST per effettuare operazioni di lettura.

Qui segue la trattazione relativa ad alcuni aspetti significativi dell'implementazione di questo modulo:

Gestione della comunicazione cross origin Il design del sistema complessivo prevede il coinvolgimento di due nomi di dominio differenti. Quando i browser eseguono delle chiamate AJAX in questo contesto *CORS* (1.4.3) eseguono una ulteriore chiamata detta di *preflight* per ottenere dal server l'autorizzazione a procedere.

Affinché il browser decida di continuare (eseguendo la chiamata effettiva) occorre che il web service abbia gestito la richiesta preliminare (di tipo `OPTION`) inviando una response accompagnata da opportuni *headers*.

```
1  protected function sendAccessControlHeaders(){
2    header("Access-Control-Allow-Origin: " .
3        "https://app.il730.online");
4    header("Access-Control-Allow-Headers: " .
5        "X-Auth-Token, X-Requested-With, Content-Type, Accept");
6    header("Access-Control-Allow-Methods: " .
7        "PUT, DELETE, POST, GET, OPTIONS')
8  }
```

Listing 2.12: Invio degli headers di risposta per la richiesta di *preflight*.

Gestione della sessione utente Ogni richiesta del client è accompagnata da un header `HTTP_X_AUTH_TOKEN` attraverso il quale viene allegato un `JSON Web Token` (1.4.2.2) ricevuto precedentemente come esito del processo di login.

In una fase preliminare a quella di attivazione dello specifico controller connesso alla richiesta, il token viene decodificato impiegando la libreria `Firebase` e la chiave segreta configurata sul server.

Nel caso in cui il token risulti valido, non scaduto e con a bordo un *claim* idoneo ad identificare un utente del sistema, il modulo API provvederà ad assegnare l'oggetto *user* dell'applicazione con una concretizzazione dell'interfaccia `IWebUser`[34].

Da quel momento in avanti, il controllo passerà dal framework alla specifica action identificata dalla richiesta e la sua implementazione potrà far fede a tale oggetto user ignorando come la sessione sia stata ristabilita.

Il vantaggio di questo approccio è che le implementazioni dei controller sono disaccoppiate dal meccanismo di autenticazione & recupero della sessione, permettendo ad esempio di sostituirlo nel caso dovesse risultare vulnerabile.

[34] *IWebUser Interface* https://www.yiiframework.com/doc/api/1.1/IWebUser

Gestione delle eccezioni & codici HTTP Il protocollo prevede che ogni risposta sia accompagnata da uno *status code*[35] che il client valuterà per determinare l'esito del processamento della sua richiesta. É importante che dal lato server vengano emesse risposte accompagnate da codici coerenti con quanto avvenuto.

Per garantire il rispetto di questa specifica, durante l'attivazione del modulo API è stata configurata l'applicazione per usare come *errorHandler*[36] un oggetto che facesse un trattamento particolare delle eccezioni di tipo `CHttpException`[37].

Questo error handler riconduce le eccezioni generiche PHP (classe base `Exception`) ad uno status code `500 Internal Server Error` mentre quelle specifiche allo status code veicolato dall'eccezione.

In questo modo è stato possibile fornire sempre una risposta al client **evitando eccezioni non gestite ma al contempo consentendo ai client di esprimere un significato specifico nella risposta** (ad es. un tentativo di login fallito solleva una CHttpException opportunamente parametrizzata per essere ricondotta ad uno status code `401 Unauthorized` mentre un errore di scrittura al db solleva una exception generica che sarà ricondotta ad un errore 500).

2.3.2.5 Implementazione del modulo Dashboard

Questo modulo implementa l'intero backoffice di amministrazione in dotazione ad Amministratori, Supervisori ed Operatori.

Lo sviluppo di questa parte non è stato svolto da me, pertanto non verrà trattato in questa tesi.

[35] *HTTP Status Codes* https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html
[36] *CErrorHandler* https://www.yiiframework.com/doc/api/1.1/CErrorHandler
[37] *CHttpException* https://www.yiiframework.com/doc/api/1.1/CHttpException

2.4 Testing

2.4.1 Considerazioni generali

In questo capitolo verranno discusse le motivazioni che ci hanno spinto ad adottare vari metodi di testing automatizzato ed i loro ambiti di impiego.

2.4.1.1 Testing come strumento di debugging

Durante la fase di sviluppo occorre eseguire certi passaggi anche diverse decine di volte in modo da vedere ripetutamente il comportamento del codice scritto ed aggiustare progressivamente l'implementazione fino al raggiungimento di quello atteso.

Nel contesto delle SPA, il solo fatto di *raggiungere* il passaggio sottoposto ad esame può richiedere molti sforzi.

Nelle applicazioni di questo tipo un determinato scenario corrisponde ad una particolare configurazione dello stato interno. Salvo casi particolari, questo non può essere raggiunto in maniera diretta dallo stato di default, ma occorre **operare una serie di transizioni ammissibili**. Per ognuna di queste occorre **fornire dati in input validi** ed evitare percorsi che produrrebbero side-effects tali da inficiare sullo scenario da sottoporre ad esame.

Nel caso in cui lo scenario da verificare richieda un contesto *vergine*, bisogna pure provvedere a **ripulire gli effetti delle attività precedenti** prima di avviare ogni processo di debugging.

Il fatto di **riequilibrare la proporzione tra attività manuali di contorno allo sviluppo e lo sviluppo stesso** è stata la motivazione principale che ci ha spinto all'adozione di procedure di test automatiche.

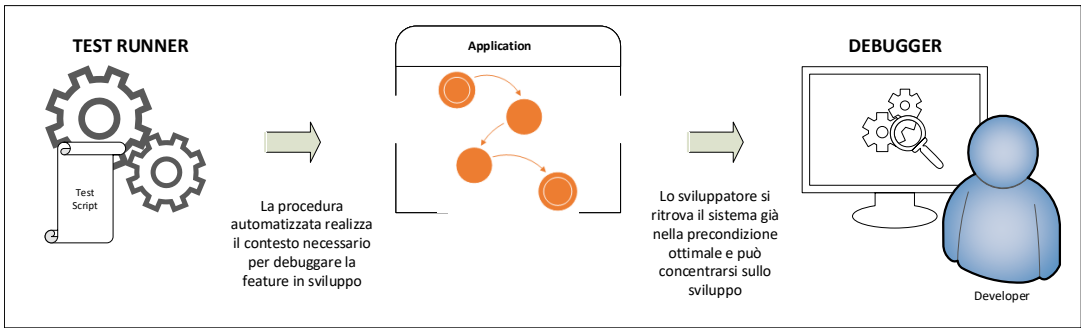

Figura 2.29: Testing automatizzato come strumento per portare in maniera efficiente il sistema nello stato necessario per portare avanti lo sviluppo.

Questa metodica permette allo sviluppatore di lavorare con più qualità perché deve concentrarsi solo sull'implementazione e non deve più interrompere il processo creativo per ricreare la condizione di cui necessita.

Il fatto di ridurre il costo della fase di debug, permette di farne un uso più frequente. In questo modo il software che esce dalla fase di sviluppo è già stato sottoposto ad un testing più estensivo e richiede meno lavoro in fase di validazione.

2.4.1.2 Testing come strumento di validazione

Testare un sistema significa **verificare che questo si comporti come atteso**. Testare ogni feature, per ogni piattaforma ad ogni rilascio diventa una procedura impossibile da portare avanti manualmente.

Per supportare un sano percorso di validazione occorre adottare test automatizzati.

L'automazione rende il costo di questa fase minimo, quindi ripetibile ogni volta che si sta per effettuare un rilascio o ogni volta che l'operato di un addetto è stato integrato con quello del resto del team.

Per quanto sia indiscutibilmente utile avere la possibilità di **verificare che i nuovi sviluppi non abbiano introdotto regressioni** e che il lavoro in team non abbia portato a **problemi di integrazione**, non è facile introdurre questa metodica.

Le procedure di test sono di fatto codice, quindi serve codice per testare il codice. Gli autori delle *suite* devono avere una perfetta conoscenza delle specifiche e ad ogni cambio di quest'ultime anche i test devono essere aggiornati.

In molti casi mettere in piedi un sistema di testing automatizzato, congetturare centinaia di scenari che mettano alla prova il software e codificarli nella forma di tests, richiede un ordine di sforzo ben maggiore di quello di tradizionale manuale.

Il vantaggio dell'approccio automatico si manifesta solo sul lungo periodo o quando il numero di features è molto alto o quando la frequenza dei rilasci è molto alta.

Il nostro progetto non ricadeva perfettamente in nessuna di queste situazioni quindi abbiamo dovuto fare un compromesso.

La scelta è stata quella di coprire con procedure di test solo le porzioni del sistema più critiche e quelle che in caso di errori non avrebbero reso il fatto particolarmente manifesto. In questo modo abbiamo aggiunto un livello di garanzia maggiore a quei componenti per cui difficilmente ci sarebbe potuta pervenire una segnalazione di bug.

In riferimento alla procedura di registrazione: se ci fosse un bug in quella fase il contribuente non avrebbe modo di segnalare il malfunzionamento (perché ancora non ha accesso ad un canale di dialogo con lo staff) e noi non avremmo modo di accorgercene perché appunto, l'operazione è avvenuta come guest e non ha lasciato impronta.

Questo ed altri **flussi chiave sono stati coperti da procedure di testing automatizzate end-2-end** in grado di simulare l'utente dalla sua registrazione al download del definitivo. Queste vengono lanciate a titolo di validazione prima di ogni rilascio importante.

2.4.1.3 Testing a titolo di specifica

I framework che abbiamo impiegato per scrivere i test abbracciano la metodologia Agile nella sua espressione *Behavior Driven Development (BDD)* [8].

L'idea di fondo è quella di riconciliare le differenze tra il team tecnico e quello di management durante il processo di sviluppo. Nonostante nel nostro caso non ci sia questa distinzione tra teams (perché la nostra è una piccola azienda) la metodica è ugualmente significativa. Lo stesso autore di una specifica potrebbe dimenticarla nel tempo ed avere uno strumento conciso che permetta di **storicizzarla e condividerla** è di valore oggettivo.

I test sono scritti in modo da ***descrivere* la feature attraverso una sequenza di affermazioni riguardo cosa *essa* deve fare**. Questi test devono intramezzare alle procedure codificate fasi descrittive che ne indichino l'intento ed il risultato atteso, in questo modo **il lettore di un test può dedurre la specifica del sistema**.

2.4.2 Mocks

Il nostro progetto coinvolge almeno 3 sistemi che comportano una interazione con *il mondo esterno*:

- Provider di invio SMS
- Gateway di pagamento bancario
- Mailer SMTP

Due di essi comportano una spesa monetaria viva quindi è impensabile prevedere di sostenere un costo ogni volta che si deve testare la procedura.

Occorre che il sistema durante la fase di sviluppo e testing sia configurato per usare implementazioni di tali servizi che non hanno risvolti penalizzanti ma che allo stesso tempo permettano al sistema di funzionare. Questo approccio prende il nome di *mocking*.

Un discorso analogo vale per le email in uscita. Sebbene queste non abbiano un impatto economico vivo, serve poter testare degli invii anche massivi, quindi non è praticabile disporre di una mole di indirizzi email privati ad uso testing.

Per mettere in atto il mocking serve un supporto progettuale. Il software deve essere scritto in maniera tale da avere un accoppiamento sulle *astrazioni* e non sulle *concretizzazioni*.

Fortunatamente abbiamo tenuto in mente questa tematica sin dall'inizio e sia la componente frontend che quella platform adottano pattern come la *Dependency Injection* e la *Dependency Fetching* che permettono di ottenere questo disaccoppiamento in maniera brillante.

I mock che abbiamo implementato rispettano i loro rispettivi contratti di servizio in questa maniera:

Mailer L'applicazione è progettata per usare un mailer esterno e non quello nativo del PHP. In questo caso il disaccoppiamento avviene a livello di protocollo (SMTP) ed il mocking si attua a livello della configurazione.

L'applicazione in produzione è configurata per usare un server di posta in uscita reale, mentre quella di sviluppo usa un mailer locale speciale che ha la caratteristica di non far uscire la mail sulla rete pubblica ma di renderla consultabile attraverso un'apposita interfaccia.

Il mailer in questione è `Mailcatcher` e permette di testare il comportamento del sistema con una fedeltà del 100% in quanto, di fatto, la mail è stata spedita dal programma; ciò che non è accaduto è solo la consegna al destinatario.

Questa soluzione ha il meraviglioso vantaggio di permettere l'uso di mail inesistenti: in questo modo è possibile simulare invii massivi, semplicemente inventando centinaia di indirizzi fasulli.

Servizio SMS L'applicazione usa gli SMS come soluzione per recapitare all'utente una password temporanea.

L'implementazione mock del servizio non effettua alcuna spedizione. Come può dunque il tester superare quelle fasi in cui è stata generata una password e lui non l'ha potuta ricevere?

Il trucco sta nel fatto che l'applicazione configurata per il testing, mocka anche il generatore di passoword!

Il generatore di password mock genera sempre la stessa password, in questo modo il tester sa già cosa inserire e non serve che gli venga recapitato alcun SMS fisico.

Payment Gateway Il sistema di pagamento è esterno e l'interazione con esso consiste sostanzialmente in un redirect che decreta l'abbandono dell'applicazione e l'approdo su di esso.

Il mocking avviene cambiando l'URL sul quale si viene ridirezionati.

Il sistema di pagamento è già progettato per supportare i test e dispone di una versione identica a quella di produzione ma operante in modalità *sandboxed*.

Dentro di essa si possono usare carte di credito fasulle per simulare tutti gli esiti di pagamento (carta scaduta, credito insufficiente, errore del sistema eccetera)

In questo modo semplicemente configurando l'applicazione per portare l'utente ad un URL diverso e programmando i test per usare speciali carte di credito è possibile ricreare qualsiasi situazione.

2.4.3 Testing app

L'applicazione pensata per i contribuenti è composta da una miriade di componenti con comportamenti rudimentali. É difficile pensare che possano nascondersi bug significativi a quel livello e operare un *test coverage* massivo su tutte quelle unità richiederebbe uno sforzo considerevole per risultati di dubbio interesse.

Per questo motivo non abbiamo scritto procedure di unit testing per quanto riguarda il frontend utente.

Il rischio che si annida invece in un sistema composto da una moltitudine di sotto elementi sta proprio nell'integrazione di tutti questi.

Trattandosi inoltre di una SPA, ricordiamo che lo stato del sistema viene maturato come una progressione di transizioni elementari: **potrebbe bastare una sola transizione non funzionante per rendere irraggiungibile un'intera porzione del grafo dell'applicazione**.

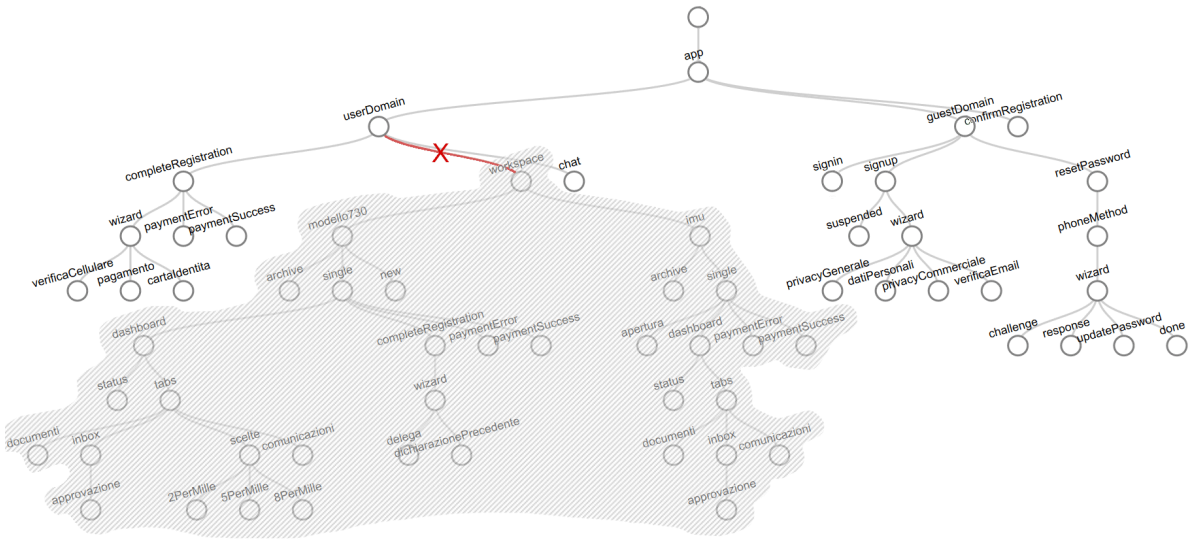

Figura 2.30: Scenario di esempio che mostra come in una SPA un bug isolato possa avere risvolti molto penalizzanti sulla navigabilità del sistema.

Per questo motivo tutti i flussi critici dell'applicazione (iscrizione, apertura di una pratica, pagamenti eccetera) **sono stati coperti da procedure di testing end-2-end**.

Per scrivere tests di questo tipo serve uno strumento che permetta di eseguire delle interazioni su un browser simulando il comportamento di un utente. Questo strumento esiste e si chiama `Selenium`. Affinché questo possa dialogare correttamente con i browser installati sul sistema occorre installare appositi moduli di interfacciamento. Fortunatamente nell'ecosistema Node.js esiste il tool `webdriver-manager` per automatizzare queste procedure di setup.

```
#Configura l'ambiente in base ai browser disponibili
webdriver-manager update

#Avvia il server Selenium
webdriver-manager start
```

Listing 2.13: Setup ambiente di esecuzione tests End 2 End.

Una volta che Selenium è in funzione possiamo cominciare ad inviare comandi ai browser. Tra i vari framework di testing disponibili abbiamo scelto `Protractor` perché è quello che meglio si integra con la tecnologia usata dalla nostra applicazione.

Una modifica nello *scope* di un'applicazione AngularJS si riflette sull'interfaccia grafica solo dopo che il framework ha compiuto il cosiddétto *ciclo di digest*. Se implementassimo dei test senza tenere conto di questo fatto correremmo il rischio di verificare l'aspettativa prima che la view dell'applicazione abbia avuto l'occasione di aggiornarsi. Questo porterebbe ad avere dei risultati falsati.

Fortunatamente Protractor adotta un protocollo di comunicazione con Selenium che implementa una *gestione delle attese* tra un comando e l'altro, rendendo questa operazione completamente trasparente agli occhi dell'autore della routine di test.

Il flusso di esecuzione di una procedura e2e è il seguente:

- Protractor viene avviato specificando a linea di comando qual'è la suite da eseguire.
- Si connette con l'istanza in esecuzione del server Selenium e richiede l'apertura di un browser.
- Le varie direttive elencate nella suite vengono tradotte in comandi WebDriver ed impartiti a Selenium per *remotare* il browser. Tipicamente la prima direttiva è quella per portare il browser sull'URL dell'applicazione.
- Protractor riconosce che quanto in esecuzione è basato su AngularJS ed inizia a sincronizzare i comandi in funzione del ciclo di digest dell'applicazione.
- L'applicazione in esecuzione nel browser remotato reagisce ai comandi ricevuti come se si trovasse di fronte ad un utente e produce gli output previsti.
- La suite di test deve essere programmata per avere delle *aspettative* e per verificarle cercando la presenza di certi elementi del DOM.

  Ricordiamo che i test e2e emulano il comportamento di un utente, quindi l'unico modo per percepire che sia avvenuto un certo accadimento è quello di *vedere* nella pagina qualcosa che faccia pensare che così sia stato.
- A questo punto viene fatto un matching tra quanto atteso e quanto verificato, proponendo un report nella console dalla quale si ha lanciato il comando.

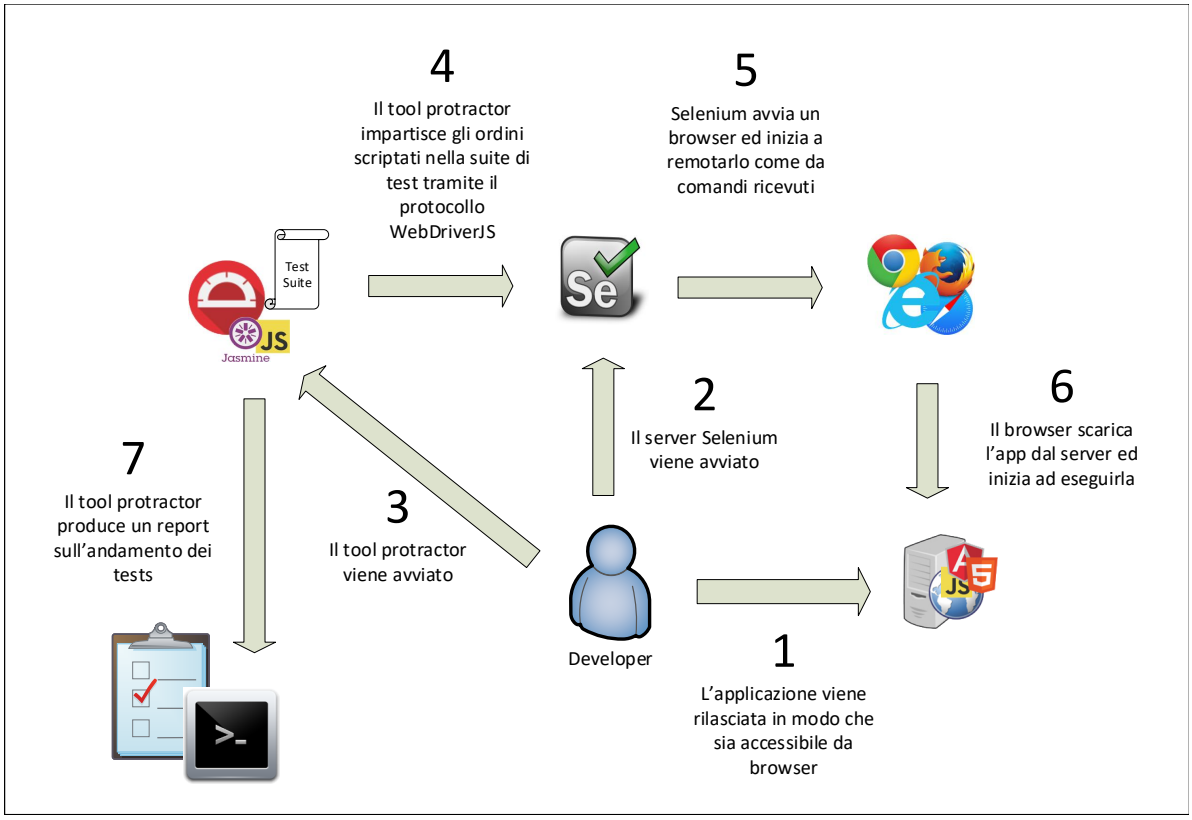

Figura 2.31: Dinamica di esecuzione dei test end-2-end con Protractor.

Come accennato nelle discussioni precedenti la stesura di procedure di test che emulano il comportamento di un utente è una problematica non affatto banale e serve un'ingegneria su vari livelli.

- Vista la necessità di interagire con il DOM dell'applicazione serve avere una visione di più alto livello, altrimenti i tests sarebbero troppo accoppiati con il markup delle pagine e al loro minimo cambiamento sarebbero completamente da riscrivere. Abbiamo adottato il pattern *Page Object*
- Diverse azioni elementari dell'utente si espletano attraverso numerose attività di basso livello.

  Ad esempio l'operazione *effettua il login* si traduce in:

  - Naviga fino alla pagina di login.
  - Porta il focus sul campo user name.
  - Digita l'username.
  - Porta il focus sul campo password.
  - Digita la password.
  - Scorri la pagina fino a far comparire nello schermo il bottone di login.
  - Clicca sul bottone login.

  Queste operazioni basilari si ripetono centinaia di volte quindi serve un metodo per poterle riusare senza doverle riscrivere.

  Abbiamo adottato il pattern *Step Object*.
- I test devono essere *parlanti* in modo che leggendoli si possa dedurre la specifica.

  I framework Protractor e Jasmine hanno delle direttive (*describe*, *it*, *beforeEach*) che se usate opportunamente permettono di raggiungere questo scopo.

  La condotta da tenere nella stesura dei test è a grandi linee questa:

  - Usare frasi al verbo gerundio per il blocco *describe* (es. 'andando', 'loggando', 'cliccando' eccetera).
  - Implementare il codice che attua lo scenario *descritto* nell'handler *beforeEach*.
  - Usare frasi assertive al verbo presente per il blocco *it* (es 'accade questo e quello').
  - Implementare nel blocco *it* la verifica sull'aspettativa indicata nel titolo usando le API del framework Jasmine.

```
1
2  describe("Andando alla pagina di Login", function(){
3
4    beforeEach(function(){
5      //Uso dello StepOpject
6      guestUser.goToLoginPage();
7    });
8
9
10   describe("e tentando un accesso con dati fasulli", function(){
11
12     beforeEach(function(){
13       //Uso del PageObject
14       loginPage.fillForm("verryFakeUzer", "s3cret");
15     });
16
17     it("il login viene respinto", function(){
18
19       //Uso dell'oggetto 'browser' offerto da Protractor
20       //Uso delle API expect-to di Jasmine
21       expect(browser.getCurrentUrl()).toBe('/accessDenied');
22     });
23   });
24
25
26   describe("ed inserendo credenziali valide", function(){
27
28     beforeEach(function(){
29       loginPage.fillForm("goodBoy", "safePassword");
30     });
31
32     it("il login avviene con successo", function(){
33
34       expect(browser.getCurrentUrl()).toBe('/welcome');
35
36       //Uso dell'astrazione sul DOM operata a livello del PageObject
37       //insieme ai metodi di protractor (isXXX)
38       //ed alle aspettative (come usuale)
39       expect(welcomePage.saluteMessage.isDisplayed()).toBeTruthy();
40     });
41   });
42 });
```

Listing 2.14: Specifica di esempio tramite test e2e.

La precedente *spec* contiene due affermazioni (i blocchi it) quindi produrrà a video 2 messaggi. Tali aspettative saranno verificate dopo aver completato l'esecuzione della cascata di blocchi beforeEach.

Il pattern precedentemente descritto è stato rispettato in quanto i blocchi it contengono solo affermazioni ed aspettative, mentre i blocchi beforeEach contengono solo azioni.

In alcuni testi che reputo di buona fattura (es. [11] e [8]) veniva usato uno stile verbale differente. A mio avviso questo che ho descritto è il migliore perché permette veramente di avere dei dei test che han valore di specifica.

```
1  OK - "Andando alla pagina di Login e tentando un accesso con dati
       fasulli il login viene respinto"
2
3  OK - "Andando alla pagina di Login ed inserendo credenziali valide il
        login avviene con successo"
```

Listing 2.15: L'output dell'esecuzione della procedura 2.14 mostra *la specifica*.

2.4.4 Testing platform

Per la componente lato server è risultata necessaria una copertura con i test molto inferiore rispetto a quella richiesta per l'app.

Il modulo API viene già implicitamente testato nel momento in cui si eseguono i test e2e sull'applicazione, quindi non richiede ulteriore copertura.

Il modulo dashboard (che realizza il backoffice di amministrazione) esegue operazioni CRUD elementari basate per lo più su codice del framework Yii, quindi ha una probabilità bassa di ospitare bug importanti. Inoltre è usato da un numero limitato di addetti che hanno con noi un rapporto di collaborazione (sono loro che danno l'assistenza tecnica di primo livello ai loro clienti e ci trasmettono le richieste di feature e le segnalazioni di bug). Per questo motivo il modulo dashboard non è stato coperto da procedure di test.

Le uniche procedure di test che sono state sviluppate in questo ambito coprono i componenti interni più avanzati.

Trattandosi di entità usate via API interna, **non c'era alternativa al loro testing se non per via programmatica**.

Inoltre questi componenti incorporano un discreto **livello di complessità** e **svariati possibili impieghi**: l'unico modo per garantire la qualità dell'implementazione era passando per un'accurata fase di *unit testing*.

I componenti interessati da questo tipo di copertura sono:

Motore di notifica Ogni partecipante di una chatroom può avere visto i messaggi per l'ultima volta in momenti differenti, quindi ognuno ha un suo conteggio di notifiche personalizzato.

I messaggi scritti dall'autore non cumulano notifiche per lui ma solo per gli altri partecipanti alla chat.

I messaggi rimasti non letti per un certo lasso di tempo vengono segnalati via email.

Questi spunti sono sufficienti a far capire che il componente deve fronteggiare una miriade di casi limite che non potrebbero essere testati manualmente con un buon livello di affidabilità.

Sono dunque stati scritti degli **Unit Test per validare la qualità dello sviluppo fatto**. L'implementazione è risultata così buona che non sono mai emersi dei bug e di fatto quel codice non è stato più toccato.

Motore di messaggistica La trattazione fatta per le notifiche và a braccetto con quella dei messaggi, in quanto *la sostanza* che li governa è la stessa.

Anche per questo componente abbiamo preferito avere una buona certezza sulla qualità del lavoro svolto ed abbiamo scritto dei test.

Validatore delle password Tra i vari requisiti di rigore imposti per il sistema c'era quello di adottare una password compliant con la normativa in vigore. Per l'occasione abbiamo dovuto scrivere una regular expression dedicata ed abbiamo preferito assicurarci che fosse ben fatta sottoponendola a numerosi input limite.

Sostanzialmente non ci sono stati altri casi di componenti che richiedessero un testing così approfondito.

```
class SampleNotificationTest extends CTestCase{

  /*
    PRE CONTESTO

    aliasCanale   aliasSottoscrittorePusher payload

    canale_target sottoscrittore_extra1   ONE
    canale_extra1 sottoscrittore_extra1   TWO
    canale_target sottoscrittore_extra2   THREE
    canale_extra2 sottoscrittore_extra1   FOUR
    canale_target sottoscrittore_extra1   FIVE
    canale_extra2 sottoscrittore_target   SIX
    canale_target sottoscrittore_target   SEVEN
    canale_extra1 sottoscrittore_extra1   EIGHT
    canale_target sottoscrittore_extra2   NINE
    canale_extra1 sottoscrittore_target   TEN

  */


  public function testTake(){


    //Prendi tutte le notifiche per il contribuente 'target'
    //relativamente al canale di notifica
    //collegato alla chat room del contribuente 'target'
    $allNotifications = Notifications::service()
        ->impersonate('contribuente_target')
        ->accessChatOf('contribuente_target')
        ->take();

    $this->assertCount(
        4,
        $allNotifications,
        "'canale_target' contiene 5 notifiche di cui 1 emessa dal
           soggetto");

    $this->assertEquals(
        "ONE THREE FIVE NINE",
        Utils::implodeNotificationsAsString($allNotifications),
        "Take deve ritornare solo le notifiche non emesse dal soggetto
           in quel canale");



```

```
46      //Ripetendo la stessa operazione
47      $allNotifications2 = Notifications::service()
48          ->impersonate('contribuente_target')
49          ->accessChatOf('contribuente_target')
50          ->take();
51
52      $this->assertCount(
53          0,
54          $allNotifications2,
55          "L'invocazione precedente di take deve aver segnato tutte come
                lette");
56    }
57  }
```

Listing 2.16: Test case di esempio.

3
Dev Ops

Negli ultimi anni la tecnologia web sta rivestendo un ruolo di importanza strategica in continuo aumento. Sono sempre di più le aziende che stanno abbandonando lo sviluppo su piattaforme native in favore di soluzioni puramente online, questo per migliorare la produttività e per fornire velocemente applicazioni incredibilmente complesse ai loro clienti [15].

Questo fenomeno sta portando nelle web agencies più storiche un nuovo problema da affrontare. Sin dagli albori del web, il core business di queste aziende consisteva nella realizzazione di siti Internet e/o la gestione di campagne di promozione online. Questo tipo di servizi aveva una connotazione prettamente visuale e di marketing, quindi approcci ingegneristici allo sviluppo del servizio non risultavano necessari.

Oggi invece, per cavalcare questo fenomeno di abbandono della piattaforma desktop, hanno dovuto introdurre nel loro organico figure professionali nuove. Questo porta ad una spaccatura interna in quanto il personale storico, difficilmente riesce a mettersi al passo in tempi brevi e con costi per l'azienda accettabili.

Il rischio maggiore è quello di innescare un fenomeno di stallo in cui le nuove risorse invece di esprimere il loro potenziale nell'introduzione di novità, vengono frenate nel tentativo di trascinare e convertire il *legacy aziendale*. Allo stesso tempo però bisogna evitare, o comunque limitare, il fatto che la conoscenza sui nuovi progetti divenga a solo appannaggio di pochi individui.

In uno scenario del genere diventa di fondamentale importanza l'introduzione di **metodiche di lavoro aziendali** che permettano di abbracciare la novità, frammentando il meno possibile il team, limitando i costi per la loro adozione ed evitando rischi d'impresa.

Nelle sezioni che seguiranno verrà raccontato come sono stati affrontati i problemi organizzativi di questo progetto e di come le soluzioni elaborate abbiano segnato una **tappa nell'evoluzione aziendale**.

3.1 Il profilo aziendale e la genesi del nuovo workflow

L'azienda in cui è stato sviluppato questo progetto esiste da molti anni e nel corso della sua storia ha portato alla luce **numerosi sistemi di grandi dimensioni** operanti negli ambiti più disparati: dalla sanità, alle amministrazioni comunali, alle fondazioni no profit di scala internazionale.

Tutti questi progetti sono accomunati dal fatto di esser stati sviluppati da **micro team composti da pochissime persone**.

Benché il fatto che non servano molte risorse ma solo *quelle giuste* sia interessante dal punto di vista economico, questo ha dei risvolti meno positivi dal punto di vista organizzativo.

Centralizzare tante mansioni su pochi soggetti porta alla comparsa di individualismi e questo spesso confligge con l'intento della direzione di **avere uno standard di produzione** aziendale. Anche la tematica della formazione interna và di pari passo con questo fenomeno.

Difficilmente risulta possibile distogliere svariati soggetti dalle mansioni operative, perché in team piccoli vorrebbe dire avere un fermo di produzione. Ciò che ne deriva è che le competenze si *verticalizzano* e per ogni progetto si finisce per avere una sorta di figura *guru* ed una mole di informazioni non scritte e non condivise.

Questo particolare progetto richiedeva una totale padronanza del linguaggio JavaScript ed anche una grande disciplina nel suo uso, in quanto è un linguaggio concepito per fare dello scripting che non offre nessuno strumento per dominare la complessità collegata ad un suo uso massivo.

I progetti storici fatti in azienda avevano sempre avuto una forte componente di programmazione lato server in PHP oppure in ambito desktop con il linguaggio Delphi. L'esperienza del personale in materia JavaScript era dunque molto limitata e non era mai andata oltre alla scrittura di semplici routine.

Nello lo sviluppo della componente frontend de IL 730 ONLINE (circa 30K linee di codice JS) mi sono ritrovato ad affrontare un *lungo cammino solitario*.

Con un team con un background così disomogeneo e con un linguaggio così destrutturato sarebbe stata una scelta terribile quella di forzare un lavoro in team dove evidentemente non c'erano sufficienti requisiti di base per farlo.

La scelta più saggia è stata quella di dividerci le attività in base alle competenze che già ognuno possedeva.

A questo punto il problema aveva preso una connotazione nuova, ossia *come condividere la conoscenza*.

Durante questo percorso ho dovuto necessariamente introdurre l'uso di strumenti e metodi ed ho sentito presto che non solo tutto questo non era condiviso ma spesso nemmeno compreso. Gli altri membri del team ignoravano le motivazioni delle mie scelte e non sapevano quali erano gli strumenti da usare, in quali fasi del progetto erano necessari e come andavano impiegati.

Era evidente che serviva un qualche sistema per ridurre questo divario che fosse **più forte di un documento di istruzioni**.

Un pò per lungimiranza ed un pò per fortuna, mi sono ritrovato ad aver scelto delle soluzioni che oltre a risolvere il problema operativo del momento, con un pò di sforzo ulteriore potevano diventare una **soluzione per decentralizzare dall'individuo lo svolgimento delle operazioni e la conoscenza che esse richiedevano**.

Mi riferisco in particolare all'**impiego dell'automazione**. Introdotta inizialmente come strumento a supporto per lo svolgimento di task ripetitivi è diventata ben presto una filosofia progettuale.

Il fatto di **codificare qualsiasi processo che richieda una forma di conoscenza** porta innumerevoli vantaggi:

- L'attività svolta da un automa avviene con grande velocità, precisione e ripetibilità.
- La procedura scriptata è parte integrante del progetto stesso, quindi non esiste nessuna forma di sapere (relativa ai processi) non condivisa.
- La procedura scriptata è aperta, quindi chi è interessato può studiarla per comprendere cosa avviene realmente, propagando la conoscenza.
- La procedura è accessibile a tutti, quindi anche chi non ha la minima idea di cosa faccia e di come lo faccia è in grado raggiungere l'obiettivo eseguendo *le operazioni*.
- La procedura non indica solo il *come* viene fatta l'attività ma anche *qual'è lo scopo* quindi se in futuro non dovesse risultare più idonea può essere aggiustata senza ricominciare da capo. É dunque manutenibile.
- Sul lungo periodo le problematiche dei progetti diventano ricorrenti. Queste metodiche possono essere riusate portando efficientamento ed omogeneità.

Il frutto di questo studio è risultato così valevole che il nuovo metodo è stato adottato almeno in parte per ogni progetto iniziato in seguito e *retrofittato* per quelli precedenti più meritevoli.

3.2 Panoramica del workflow

Il concetto alla base è quello **di evitare by design l'esistenza di forme di conoscenza che risiedano in un luogo diverso da quello del repository del progetto**.

Sostanzialmente anche *l'ultimo arrivato* una volta ricevuto l'accesso ai sorgenti deve poter accedere a tutte le informazioni essenziali per la gestione del ciclo di vita del progetto senza doversi far affiancare da un qualche esperto.

Ogni forma di divulgazione deve essere messa in campo con un approccio difensivo che non si basa sulla speranza di vedere onorate le guidelines da parte degli interessati ma di forzarne il rispetto adottando ogni strumento possibile. Questo per limitare gli errori di interpretazione e la creatività degli individui più pigri.

Sostanzialmente significa **rendere automatizzate le principali fasi di un progetto** quali ad esempio:

- Il recupero delle dipendenze.
- La compilazione dei sorgenti in un artefatto.
- Il testing.
- La posa in opera della soluzione elaborata.

Anche l'informazione su quali sono gli esecutori di queste procedure automatizzate e come devono essere configurati deve essere centralizzata allo stesso modo e richiedendo il minor lavoro manuale possibile.

Un'altra tematica degna di interesse è quella delle **autorizzazioni**. La fase di rilascio finale tipicamente richiede il possesso delle chiavi di accesso agli ambienti remoti. Questa informazione è quanto di più prezioso ci sia associato ad un progetto online! In questo caso quindi è desiderabile un processo inverso, in cui le operazioni sono ad appannaggio di tutti, ma la conoscenza ad esse associata è segreta.

Esistono poi delle informazioni che non necessariamente sono traducibili in operazioni ma che comunque è bene che siano condivise e storicizzate. Un esempio di queste sono la lista dei bug noti, le discussioni tra i membri del team e la documentazione del progetto.

L'implementazione di un workflow che soddisfacesse queste specifiche è stato reso possibile sostanzialmente da 4 strumenti fondamentali:

GitLab É la soluzione che abbiamo scelto per avere un repository remoto GIT, una gestione tipo Wiki per la documentazione, una gestione di *issues* per tracciare i bug aperti e le richieste di features ed uno strumento centralizzato per l'esecuzione di pipelines di CI/CD.

Vagrant É lo strumento che grazie all'uso della virtualizzazione permette di ricreare sulla workstation dello sviluppatore un ambiente di lavoro ottimale tagliato specificatamente per il progetto e dotato di tutti gli strumenti da esso richiesti.

Package Managers Sono usati per recuperare i tool necessari per la gestione del ciclo di vita del progetto (principalmente compilatori e strumenti di testing). Sono usati inoltre per gestire il recupero e l'installazione delle dipendenze software del progetto.

Ne sono stati coinvolti di diversi, a seconda della fase specifica del progetto e dell'ambito di impiego.

Task Runners Sono usati per eseguire delle procedure seguendo un copione scriptato (che viene versionato alla pari dei sorgenti del progetto).

Sono stati usati sostanzialmente in ogni fase del progetto dalla compilazione, al testing, al build ed al rilascio.

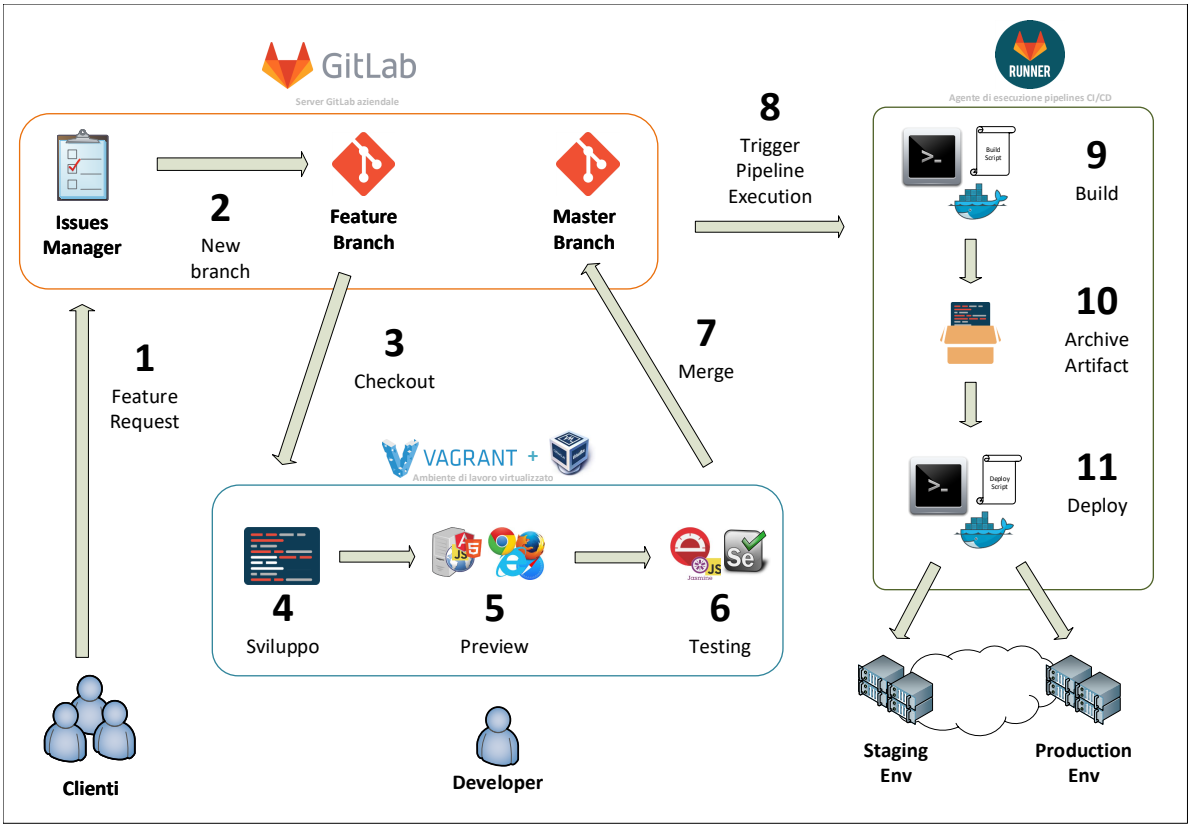

Figura 3.1: Il nostro workflow di produzione del software.

1. I clienti segnalano attraverso la chat con l'operatore **bug e richieste di nuove implementazioni**. Queste vengono inserite nella sezione Issues di GitLab in modo da conservarne uno storico ed organizzare il lavoro in base alle loro priorità. In seguito viene fatta una valutazione sulla riproducibilità dello scenario descritto, il fatto che rientri o no nell'assistenza ordinaria e se vi sia budget a sufficienza per gestirle. L'interfaccia permette di inserire commenti, referenziare snippet di codice ed eventualmente iniziare lo sviluppo della patch creando un branch dedicato.

2. Lo sviluppo dell'attività non deve mai avvenire sul branch master se non in rarissimi casi in cui si è certi della velocità dello sviluppo e del fatto che questo non introduca bug. Come linea di principio generale **ogni nuovo sviluppo deve essere portato avanti su un branch dedicato** in linea con la metodologia *Feature Branch* (1.1.2.3)

3. Lo sviluppatore prepara la sua working copy locale per avviare il processo di sviluppo. I sorgenti recuperati contengono un file descriptor (`vagrantfile`) che processato dal tool `Vagrant` permette di creare un ambiente di lavoro virtualizzato

basato su `Virtual Box` che rispecchia l'ambiente in cui verrà rilasciato il sistema una volta completato. Questo ambiente ha a bordo tutti gli strumenti necessari per proporre un'esperienza di sviluppo efficiente.

4. Lo sviluppatore **modifica i sorgenti** operando dal sistema operativo *host* usando il suo IDE di preferenza. Noi abbiamo usato NetBeans.

5. Il sistema Vagrant esegue in background un *network share* tra macchina fisica e quella virtualizzata in modo che i sorgenti siano automaticamente accessibili anche da dentro la macchina *guest*. Grazie ai server ed alle configurazioni fatte in fase di creazione di tale ambiente è possibile **eseguire il sistema in anteprima** con un apprezzabile livello di fedeltà visto che la macchina è stata appositamente preparata per rassomigliare il più possibile a quella di produzione.

6. Allo stesso modo con cui sono stati *provisionati* i servizi di sistema che permettono l'esecuzione del progetto, a bordo della macchina guest sono disponibili i framework e gli strumenti necessari per **eseguire le procedure di test**.

   Abbiamo scelto di eseguire le procedure manualmente in questa fase e non automaticamente prima del rilascio per poter avere rilasci immediati (a volte siamo certi che la modifica introdotta non può introdurre regressioni e sarebbe penalizzante dover aspettare l'intera esecuzione del ciclo prima di andare online).

   L'altro motivo che ci ha spinto ad agire in questa fase è che *è difficile* scrivere procedure di test perfette. Alcune (rare) volte accade che il test fallisca per un particolare *glitch* avvenuto durante la sua esecuzione senza che a questo episodio vi sia un bug reale alle spalle. **Volevamo evitare che un falso negativo prevenisse il rilascio** quindi abbiamo deciso di gestire la fase di test manualmente.

7. Una volta che il lavoro sul feature branch è completo e testato può essere fuso con quello sul master.

   Nel caso lo sviluppo della feature si sia protratto per del tempo e nel frattempo siano state introdotte nuove commit sul master, era a carico dello sviluppatore eseguire delle *pull* periodiche in modo da mantenere i due filoni progettuali il meno discostanti possibile.

   Quando questo metodo viene eseguito con diligenza il merge è quasi sempre una operazione indolore.

8. I sorgenti del progetto contengono un file descrittore (`.gitlab-ci.yml`) che istruisce il gestore delle pipelines di CI/CD di GitLab su come comportarsi a fronte della comparsa di una nuova versione sul repository.

   Le nostre pipelines sono progettate per avviarsi ad ogni merge sul filone master quindi l'operazione appena fatta scatena questa esecuzione.

In particolare l'esecuzione viene portata avanti da un agente detto *runner* che si avvale della tecnologia della *containerizzazione* per **eseguire le procedure in isolamento e senza lasciare *scorie* sul sistema**.

9. Il runner istanzia un container di base idoneo per il task che deve eseguire. Ad esempio per il repository 'app' viene selezionato un container che ha già a bordo il motore di esecuzione `Node.js`; vengono quindi recuperati tutti i tool necessari per la **fase di build** (tramite il suo package manager `npm`) ed avviata la produzione dell'artefatto con un task anch'esso scriptato insieme ai sorgenti.

10. Il frutto della fase di build viene collezionato dal sistema per 2 scopi: (a) Rendere l'artefatto disponibile in altre fasi (es. quella di deployment) (b) Rendere disponibile l'artefatto attraverso l'interfaccia web di GitLab per un eventuale ispezione manuale e per storicizzare le varie release.

11. Ora viene nuovamente istanziato un container che ha gli strumenti a bordo specifici per effettuare il **rilascio sulla piattaforma remota**.

    Esistono diverse soluzioni per fare un versamento di file su un host remoto, quali ad esempio `FTP`, `RSYNC` o `SSH`.

    Questi tool vengono scelti in base al fatto che serva o no eseguire anche delle *procedure di maintenance* o sia sufficiente solo fare un trasferimento di file.

    Le credenziali necessarie per la connessione vengono impostate tramite l'interfaccia web di GitLab ed iniettate a runtime dentro al runner sottoforma di *variabili d'ambiente*.

    In questo modo **gli script di rilascio (che sono versionati tra i sorgenti) possono essere divulgati perché non contengono informazioni sensibili**.

    Le nostre pipelines sono programmate per effettuare il deployment automaticamente sull'ambiente di staging, mentre su quello di produzione bisogna entrare nell'interfaccia web di GitLab ed avviare manualmente l'attività.

    In questo modo ad ogni merge troviamo sull'ambiente di staging la versione che sta per diventare quella futura ed abbiamo modo di effettuare eventuali ultime prove prima di procedere col rilascio sull'ambiente pubblico.

3.3 Teamworking

3.3.1 Adozione di un server GitLab aziendale

Tradizionalmente in azienda i sorgenti sono stati versionati con la tecnologia `SVN` o con `GIT`.

SVN è una soluzione esclusivamente client-server, quindi non può essere impiegata senza disporre di una opportuna infrastruttura di rete. Inoltre è una soluzione più datata che sta lentamente venendo soppiantata per via dell'esistenza di alternative che hanno un corredo di features migliore ed un parco di software più completo. Anche noi abbiamo seguito questo trend e da diversi anni a questa parte usiamo esclusivamente GIT.

GIT permette di operare in maniera *serverless*, versionando i sorgenti su un repository locale che può funzionare perfettamente anche in assenza di rete. Questa copia locale può essere allineata periodicamente con una remota (detta *origin*) allo lo scopo di mantenere un backup ed avere un luogo centralizzato con cui condividere il lavoro con il team.

L'azienda non disponeva di un server GIT interno e non ha mai voluto ospitare i sorgenti presso provider esterni (quali ad esempio GitHub, BitBucket) quindi questi venivano conservati in una NAS (Network Area Storage).

Benché questa soluzione sia perfettamente funzionante dal punto di vista tecnico non è ottimale.

Quando si accede al repository tramite una *network share* il sistema operativo mette in atto un'astrazione per mascherare l'effettiva collocazione sulla rete rendendola accessibile con un normale *path* di sistema. In un contesto del genere, il software Git *pensa* di disporre dei sorgenti *in locale* ed opera un accesso al file system aggressivo. Questa mole di `I/O` quando viene veicolata sulla rete ne mette in evidenza i limiti. In particolare la latenza diventa una metrica non più trascurabile ed in particolari condizioni può risultare addirittura inaccettabile.

Per questo progetto capitò spesso di dover lavorare fuori dall'ufficio tramite una *VPN* su rete geografica pubblica. Durante le operazioni sul repository più onerose (es. merge) capitava che queste andassero in *timeout* per via dell'inadeguatezza del canale di comunicazione rapportato alla policy di accesso ai files usata dal client git.

É stata questa la motivazione principale che ci ha spinto a **dotare l'infrastruttura di rete aziendale di un server GIT**.

Tra le varie soluzioni installabili in maniera standalone abbiamo scelto GitLab[1] perché dimostrava di avere una certa robustezza, era free (e lo è ancora in parte al momento della presentazione di questa tesi) ed è ben supportato. Questo oltre ad offire la funzionalità core che cercavamo (cioè la gestione di un server GIT remoto) offre altri strumenti interessanti utili nell'intero ciclo di sviluppo software.

[1] *GitLab* https://about.gitlab.com/

Le principali funzioni offerte da GitLab sono:

- Un'interfaccia web ben fatta per l'amministrazione dei repository e degli utenti.
- Un sistema basilare di gestione di *issues*. Nonostante questo non abbia tutte le features di un sistema di *ticketing support* è comunque un valido strumento perché permette di fare delle annotazioni che referenziano i sorgenti e di derivare direttamente un branch dove ospitare gli sviluppi associati alla issue.
- Ha una gestione interna di Wiki, che permette di creare un mini sito di progetto dove inserire le informazioni di partenza necessarie per gli sviluppatori per cominciare a contribuire. Questo viene generato automaticamente dal sistema parsando i files scritti in linguaggio *Markdown*[2] versionati insieme ai files del progetto.
- Offre funzioni di Continuous Integration/Continuous Deployment.
  Queste inizialmente non sono state una metrica che ci ha fatto preferire GitLab ad altre soluzioni ma *col senno del poi* possiamo dire che è risultato estremamente vantaggioso avere un'unica soluzione centralizzata. Se avessimo scelto un gestore di repository diverso tra quelli disponibili sul mercato (es. *BitBucket*, *GitBucket*, *Gogs*) nessuno di queste avrebbe offerto funzionalità di DevOps e sarebbe risultato necessario dover attivare un altro server con un software specifico (quale ad esempio *Jenkins*). Questo avrebbe richiesto una laboriosa fase di configurazione per farli dialogare.

3.3.2 Adozione della strategia 'Feature Branch'

L'idea era quella di limitare il più possibile la comparsa di conflitti di versione da gestire. Nel caso in cui questi erano inevitabili (ad esempio perché gli stessi file erano stati modificati su 2 filoni di progetto distinti) il problema doveva comunque sollevarsi prima del momento del merge finalizzato al rilascio, cioè durante la fase di sviluppo.

Per raggiungere questo obiettivo abbiamo adottato una strategia nota in letteratura con il nome di 'Feature Branching' [5] impiegandola con approcci a corto ciclo di sviluppo tipici della metodica *eXtreme Programming*.

Le regole da rispettare sono le seguenti:

- Bisogna designare un branch per essere quello *mainline*, ossia quel branch in cui ogni commit corrisponde ad una versione stabile del sistema. Noi abbiamo scelto quello master. L'idea è che questo branch debba sempre contenere una release stabile, in modo che vi si possano stratificare sopra veloci bugfix e poter rilasciare immediatamente, senza avere di intralcio le commit relative a features incomplete (che ne impedirebbero la rilasciabilità).

[2] *Linguaggio Markdown* https://it.wikipedia.org/wiki/Markdown

- Ogni feature che prospetta una certa laboriosità deve essere necessariamente versionata su un branch diverso da quello master.

- Ogni membro attivo sullo sviluppo di una feature deve controllare spesso il master branch, *pullando* nella sua copia locale le modifiche avvenute sull'*origin* e *mergendole* sulla sua copia locale del suo branch feature. Questa operazione è nota come *rebase* ed ha lo scopo di anticipare la scoperta di eventuali conflitti di versione. Applicando questa regola con diligenza si ha il benefico effetto per il quale il branch della feature contiene sempre al suo interno la storia del master (cioè del branch con cui dovrà fare il merge una volta completata). Un merge tra 2 branch che condividono una storia comune è noto come *fast-forward* e non produce mai effetti collaterali. L'altro vantaggio è che il branch della feature, con al suo interno le versioni aggiornate del master *è* in sostanza il futuro master branch, quindi ogni test condotto su di esso è particolarmente significativo perché anticipa il funzionamento che avrà il sistema una volta in produzione.

- Ogni modifica potenzialmente impattante per il resto del team deve essere *pushata* il prima possibile in modo che il resto del team possa prendere atto di questo aggiornamento e provvedere a gestire eventuali conflitti il prima possibile. Questo approccio ha lo scopo di ridurre la comparsa o l'entità dei merge-hells. Arrivando alla realizzazione della feature principale attraverso micro sviluppi incrementali e condivisi.

Questo workflow funziona bene se tutti i membri rispettano le regole. Nel nostro caso non sono mai capitati problemi in quanto il lavoro era stato preliminarmente suddiviso in modo da ridurre le aree di intersezione. Per quelle volte in cui occorreva *fasarci* risolvevamo con un coordinamento manuale, reso possibile dal fatto che il team era particolarmente piccolo.

Per progetti in cui sono coinvolti più sviluppatori (magari distribuiti geograficamente o su diversi fusi) definire delle regole potrebbe non bastare per ottenere un coordinamento sufficiente. In quest'ottica GitLab mette a disposizione 2 strumenti.

Una è quella delle *pull request*[3]. Serve per invitare gli altri membri del team a prelevare delle modifiche che sono state pushate, senza aspettare che siano loro proattivamente a controllare il changelog e a realizzare di avere a bordo sul loro repository locale una versione obsoleta.

L'altro strumento è quello dei *protected branches*. L'operazione di push su certi branch può essere limitata in modo che questa sia consentita solo previo superamento di una serie di requisiti di sbarramento. Questi possono essere ad esempio l'ottenimento di un'approvazione da parte del responsabile di progetto o da parte di un numero minimo di altri membri del team.

[3] *Pull request* https://yangsu.github.io/pull-request-tutorial/

3.4 Setup delle Workstation

É noto che la non-qualità di un software dipende anche dalla disomogeneità degli ambienti di lavoro dei vari sviluppatori, al punto che svariati siti di settore fanno dell'humor[4] su questo fatto.

Non è solo importante avere **postazioni di lavoro livellate nelle loro dotazioni** ma è anche fondamentale che queste abbiano il più alto grado di **somiglianza con quello di produzione** in cui avverrà il rilascio. In questo modo molti potenziali bug possono essere evitati sin dalle prime fasi di sviluppo.

Questo progetto richiedeva lo sviluppo di un sistema web client-server. Occorreva quindi realizzare sulla workstation degli sviluppatori un setup che permettesse (tramite una rete) l'interazione tra il browser ed un server con a bordo i sorgenti *work in progress*.

3.4.1 Soluzioni di preview

Gli approcci tradizionali al problema sono quelli elencati qui di seguito:

3.4.1.1 Uso di un server remoto di sviluppo

La soluzione consiste nel predisporre un server aggiuntivo, clone di quello di produzione dove depositare i sorgenti in fase di sviluppo per avere un'anteprima della loro esecuzione.

PRO L'uguaglianza con il server di produzione è al 100% quindi il preview è ottimale.

PRO Essendo un server svincolato da quello di produzione può essere alterato per supportare lo sviluppatore (ad es. installando il modulo per il debugging PHP *xdebug*[5])

CONTRO Ogni sviluppatore deve poter testare l'esecuzione della sua *working copy* quindi di fatto servirebbe un server per ognuno!

CONTRO Il fatto che i sorgenti debbano essere uploadati in remoto per poter essere visti girare implica per lo sviluppatore l'uso di un IDE che abbia una funzionalità di *upload-on-save*.

PRO Permette un testing nativo con devices mobili. Disponendo di una rete wireless speciale è possibile configurare (via DHCP) questi ultimi per risolvere il dominio del progetto verso l'IP del server di sviluppo.

Questa soluzione è stata scartata in quanto l'onere di attivare N istanze del server di sviluppo era assolutamente inaccettabile.

[4] *"Sul mio computer funzionava"* https://blog.codinghorror.com/the-works-on-my-machine-certification-program/

[5] *XDebug* - Modulo PHP per l'esecuzione passo-passo https://xdebug.org/

3.4.1.2 Uso di un server localhost

L'idea è quella di installare sulla workstation di ogni sviluppatore tutti i servizi a livello di sistema necessari per il progetto e di interrogarli in locale.

Sia per Windows che per OS X esistono dei pacchetti pronti all'uso per il setup di server web basilari operanti in locale (*XAMPP*[6] e *MAMP*[7])

CONTRO La maggior parte dei server web sono derivati da Linux, mentre sulle postazioni desktop tipicamente si usa Windows o OS X.

Il fatto di far girare il server in locale dà una preview attendibile solo quando l'OS sulla postazione dello sviluppatore è della stessa famiglia di quello sull'host finale.

Decidendo di usare Linux come sistema operativo sulla workstation si perderebbe la possibilità di utilizzare sofisticati software ad interfaccia grafica (es. Adobe Photoshop, MS Excel, *Heidi SQL*[8])

Continuando invece ad usare un sistema operativo diverso non si avrebbe a disposizione un supporto idoneo per testare tutte le features (diversa organizzazione del file system, mancato supporto alle user permissions, path di sistema case sensitive eccetera).

CONTRO La workstation presto inizierebbe a pullulare di una moltitudine di software e tools rendendo impossibile il preview del progetto con le giuste versioni degli stessi.

PRO Il fatto di lavorare in locale permette di trattare una mole molto grande di dati con buona efficienza. Ad esempio ne IL 730 ONLINE abbiamo alcune tabelle SQL che superano il milione di record: testare le procedure di import con dati del genere e con una rete fisica di mezzo *si fà sentire*.

Questa è la strada che era stata sempre adottata in azienda ma puntualmente sul lungo periodo presentava grandi problemi di mantenimento (troppi software = conflitti). Inoltre il livello di preview che propone ha limitazioni intrinseche per cui non è pensabile di effettuare un rilascio scongiurando al 100% problemi dovuti alla diversità degli ambienti.

[6] *XAMPP* - Server Apache,MySQL,PHP per Windows https://www.apachefriends.org/it/index.html
[7] *MAMP* - Server Apache,MySQL,PHP per OS X https://www.mamp.info
[8] *Heidi SQL* - Tool di amministrazione database con interfaccia grafica https://www.heidisql.com/

3.4.1.3 Uso di un server dentro una Virtual Machine

L'idea è quella di sviluppare lavorando dal proprio sistema operativo di preferenza (*host*) ma ottenere una preview del sistema in sviluppo eseguendo i suoi sorgenti dentro un sistema operativo virtualizzato (*guest*) dello stesso tipo di quello richiesto dal progetto.

PRO Permette di scovare la quasi totalità dei tipici bug dovuti all'uso di sistemi operativi diversi.

PRO Permette di portare avanti più progetti facendo coesistere più versioni di software senza creare conflitti e senza *inquinare* la worksation.

PRO Permette di praticare approcci distruttivi in quanto in caso di disastri è semplice ripristinare un backup e non serve l'intervento di un sistemista.

CONTRO Occorre escogitare un sistema per portare dopo ogni modifica i file del progetto (che risiedono sulla macchina fisica host) dentro al sistema operativo guest. Esistono varie possibilità quali l'uso di protocolli di trasferimento o network share, ma ognuna presenta un compromesso da ottimizzare.

CONTRO Questa proposta (come le altre) condivide il problema per cui quando occorre fare una modifica sull'ambiente di produzione (es. l'installazione di un nuovo modulo PHP) occorre propagarla manualmente su tutte le istanze di tutti gli sviluppatori. Questa cosa oltre ad essere una attività tediosa che interferisce con il flusso naturale di sviluppo rischia di mettere in circolazione *diverse* VM di sviluppo sollevando problemi di integrazione futuri.

CONTRO L'immagine configurata specificatamente per il progetto può pesare anche diversi GB quindi non è facilmente portabile.

CONTRO Una volta costruita questa VM a distanza di tempo è difficile dire quali tool erano stati installati all'interno, con quali prassi e per quali scopi. In generale il procedimento è di difficile ripetizione se operato manualmente.

CONTRO Nel caso di doversi trovare a lavorare in una situazione di emergenza (es. da una workstation diversa, magari in viaggio) è molto laborioso ricreare il corredo minimo per eseguire il sistema.

3.4.2 Vagrant

Tra le varie possibilità elencate, quella basata sulla virtualizzazione è quella più promettente perché prende il meglio dalle altre due soluzioni.

Affiancandovi l'uso dell'automazione questa soluzione può diventare ottima.

`Vagrant` è **lo strumento che abbiamo scelto per rendere il processo di creazione della Virtual Machine uniforme, ripetibile e veloce**.

Le entità chiave sono le seguenti:

Vagrantfile É il file descrittore (scritto in linguaggio `Ruby`) che incapsula quali sono le caratteristiche che deve avere la VM e quali sono le procedure da fare per crearla ed amministrarla.

Il tool `vagrant` legge le specifiche da questo file e mette a disposizione dello sviluppatore una serie di comandi per eseguire in maniera automatizzata le procedure di *maintenance* sulla VM (creazione, prima preparazione, avvio, reboot eccetera).

Questo file deve essere distribuito insieme ai sorgenti versionandolo sul repository in modo che ogni sviluppatore coinvolto nel progetto possa processarlo sulla propria workstation per confezionare la VM necesaria per il progetto.

Ogni volta che per la soluzione sviluppata risulta necessario dotare gli ambienti di nuove features, queste devono essere codificate nel vagrantfile, in modo che ogni sviluppatore possa percepire queste modifiche ed allineare il proprio ambiente.

Provider La tecnologia vagrant permette di generare in maniera automatica delle virtual machines ma non è una tecnologia di virtualizzazione.

Per questo si appoggia ad altri servizi esterni che devono essere disponibili a livello di sistema. Attraverso dei plugin è possibile supportare qualunque provider ma nativamente è già possibile usare `Virtual Box`, `VM Ware`, `Docker` e `Hyper-V`.

Noi abbiamo scelto Virtual Box perché è una soluzione che eravamo già abituati ad usare anche per altri scopi ed ha il vantaggio di permettere l'esecuzione di qualsiasi OS (mentre ad esempio il provider Docker ha la limitazione di non permettere di eseguire OS diversi da quelli Linux-based).

Box I box sono immagini di sistemi operativi pre installati prelevabili da repository pubblici ed il vagrantfile deve indicare quale usare per inizializzare l'hard disk virtualizzato della VM al momento della sua prima creazione.

L'azienda che supporta il software Vagrant offre un catalogo di boxes ufficiale[9] contenente le principali distribuzioni libere.

[9] *Vagrant Boxes Cloud* https://app.vagrantup.com/boxes/search

Volendo è possibile caricare delle boxes personalizzate ma questo comporta il pagamento di un canone. Noi abbiamo preferito realizzare le nostre macchine partendo da una box di base ed eseguendo una routine di configurazione.

Quella che abbiamo scelto sia per l'ambito platform che per quello app è la `ubuntu/xenial64` che corrisponde ad una Ubuntu Linux a 64bit del 2016.

Script di provision Sono gli script che il tool Vagrant invocherà a macchina avviata con lo scopo di prepararla per l'uso specifico del progetto. Tipicamente è in questi script che si codificano i comandi shell per l'installazione dei software che devono essere resi disponibili a bordo della VM.

L'esecuzione di questi script può essere programmata attraverso il vagrantfile per avvenire solo durante la fase di `provision` oppure ad ogni avvio della macchina.

Ho trovato interessante l'organizzazione proposta da [16] che prevede di suddividere la procedura di provisioning in 3 script e di ospitarli in una directory `bootstrap` insieme alle risorse da essi impiegate.

Spesso è infatti indispensabile portare dentro l'ambiente virtualizzato anche specifici files di configurazione che non possono essere recuperati dalla rete come i software.

Sono un esempio di questi i *certificati SSL*, i file di configurazione dei *virtual host* di Apache ed i file di *identità SSH*. Tutto questo sempre nell'ottica di creare una Vagrant Machine il più simile possibile all'host di produzione.

I comandi della procedura di installazione sono stati suddivisi secondo questo criterio:

01-prepare-xenial64.sh Contiene i comandi di preparazione dell'ambiente, installando tutti i servizi a livello di sistema necessari.

Essendo la nostra installazione una Xenial 16.04 dispone del package manager della famiglia Ubuntu che permette di installare molto facilmente i software con il comando: `apt-get install <package-name>`

Sono un esempio di comandi eseguiti in questa fase quelli per installare a livello di sistema i server Apache e MySQL, l'interprete PHP, l'ambiente di esecuzione Node.js, i vari package manager ed i task runner che hanno una interfaccia a linea di comando.

Questo script deve essere eseguito una tantum al momento del primo avvio della macchina, detto fase di *provision*.

02-configure-app-for-xenial64 In questo script vengono eseguite le configurazioni di sistema necessarie per far funzionare l'applicazione su questa particolare distribuzione.

Sono un esempio di questi comandi la creazione del database, l'installazione dei moduli PHP necessari per supportare le librerie usate a livello software (es. SOAP), l'installazione dei certificati SSL e dei virtual hosts in Apache, la creazione di utenti di sistema ed il setup del *cron daemon*.

Anche questo script deve essere eseguito una tantum.

03-prepare-application Questo script a differenza degli altri viene configurato per essere eseguito ad ogni avvio della macchina.

In questa procedura ha senso programmare tutte quelle attività di routine da eseguire all'inizio di una sessione di lavoro.

Sono un esempio di queste il recupero delle dipendenze aggiornate tramite i package manager (NPM, Composer e Bower), l'allineamento del database in base alle *migrations* ed una prima compilazione di sorgenti (in modo che il software sia subito eseguibile in preview senza eseguire alcun comando manualmente).

Plugins Vagrant dispone di un ecosistema di plugin sviluppati dalla community per gli scopi più disparati. Quello che abbiamo trovato di fondamentale utilità è `vagrant-hostmanager`.

Essendo la nostra soluzione web based, affinché sia possibile vedere in esecuzione il sistema virtualizzato occorre mappare il dominio della web app sull'indirizzo IP locale della VM.

Questa attività può essere fatta manualmente modificando il `file hosts` di sistema così da *truccare* la risoluzione DNS per quel paricolare nome.

Il plugin sopra menzionato aggiunge questa configurazione all'avvio della macchina e la rimuove al suo spegnimento in modo da non lasciare refusi di configurazione nel sistema e di permettere di tornare ad accedere alla versione pubblica del sistema quando la VM è spenta.

3.5 Build

3.5.1 Processo di build per la componente app del sistema

L'app viene eseguita interamente dentro al browser, quindi la prima attività che avviene in background quando un visitatore naviga l'URL è il **download delle risorse** da essa richieste e referenziate tramite il file HTML.

La complessità del progetto rendeva assolutamente impossibile sviluppare l'intera soluzione in un unico file. Al momento della scrittura di questa tesi i sorgenti sono così distribuiti:

- 498 sorgenti .js
- 179 sorgenti .less
- 171 sorgenti .html

La proporzione di circa 3:1 dei sorgenti JS rispetto agli altri ha pienamente senso in quanto ad esempio ogni *stato* dell'app porta con sè un file di definizione del modulo angular, un file di definizione dello stato della SPA ed un file di definizione del componente UI.

Benché i sorgenti JS siano direttamente eseguibili nel browser sarebbe una scelta assurda quella di farli scaricare tutti sul client. L'overhead di rete dovuto alle tante connessioni sarebbe immenso ed il file index.html che referenzia i vari file richiederebbe continuamente una manutenzione per rimanere allineato ad ogni aggiunta o eliminazione.

Inoltre i sorgenti dei fogli di stile sono stati scritti con un linguaggio che non può essere interpretato nel browser, quindi richiede una *transpillazione*.

Per questi motivi è risultata indispensabile l'introduzione di una fase di build per **condensare tutti questi sorgenti** in un bundle minimale che fosse al contempo **processabile da un browser**.

Durante la fase di build vengono anche collezionate le dipendenze ottenute tramite i package manager ed organizzate in una struttura di directory compatibile con il deployment (in particolare trattandosi per la maggior parte di librerie open source volevamo evitare di distribuire nel rilascio anche i sorgenti di queste ultime).

L'implementazione è stata realizzata sotto forma di *tasks* del task runner `Gulp`. Questo è costruito per essere eseguito in un ambiente Node, quindi per la scrittura dei tasks è possibile avvalersi del vastissimo parco di strumenti disponibile sul repository `npm`.

Inoltre il meccanismo di funzionamento di Gulp è basato sulle *pipes* quindi è particolarmente indicato per codificare pipelines di trasformazione in più stadi.

La procedura di build si articola sostanzialmente in queste macro attività:

Gestione dei vendors Le librerie esterne per il progetto app sono state recuperate tramite il package manager Bower. Questo ha l'interessante caratteristica per cui ogni package esprime nel suo manifest un'informazione su quali sono i *main files* di tale pacchetto.

In questo task Gulp è stato usata la libreria `main-bower-files` per scremare questi file principali da tutti gli altri che non serve che siano inclusi del bundle (quali ad es. documentazione del package, esempi, sorgenti, test eccetera).

I files recuperati da bower vengono collezionati dentro una directory da esso amministrata nominata `bower_components`.

I vari package al suo interno presentano l'organizzazione del file system scelta dai loro rispettivi autori, quindi non seguono uno schema comune.

É inefficiente lavorare in questo modo perché per referenziare ogni libreria esterna dal file index.html bisogna onorare la sua effettiva strutturazione dentro al package, con il rischio di commettere errori.

Per dare maggiore omogeneità al tutto abbiamo usato la libreria `gulp-bower-normalize` che ristruttura i file del package in base al loro tipo, in questo modo la struttura finale della nostra cartella *vendor* è la seguente:

`vendor/<package-name>/js/<package-name>.js`
`vendor/<package-name>/js/<package-name>.min.js`

Creazione del foglio di stile Gli stili Less vengono convertiti in CSS con il tool `gulp-less`.

Le regole di applicazione dello stile seguono il criterio della *cascata* CSS, quindi all'interno di un foglio di stile possono essere inserite in qualsiasi ordine senza che il significato voluto dallo sviluppatore venga alterato.

Questo fatto è stato sfruttato per ricondurre i tanti sorgenti ad un unico foglio di stile semplicamente concatenandoli con il tool `gulp-concat`.

Infine il tutto viene minificato in modo che il file di output sia più leggero. Questo passaggio è stato implementato con il tool `gulp-cssmin` che esegue delle specifiche semplificazioni al codice senza cambiarne il funzionamento.

```
1  gulp.task('app:make_css', function(){
2
3    //Preliminary setup
4    [...]
5
6    //1 - Compile
7    .pipe(less({
8      relativeUrls: true,
9      paths: [
10       cfg.appSrcDirPath,
11       ...
12     ]}))
13
14   //2 - Condensate multiple sources into one
15   .pipe(concat('app.css'))
16
17   //3 - Minify
18   .pipe(cssmin())
19   .pipe(rename({
20     suffix: '.min'
21   }))
22
23   //4 - Output
24   .pipe(gulp.dest(cfg.cssOutputDirPath));
25 });
```

Listing 3.1: Stralcio della procedura di compilazione del CSS dell'app.

Creazione del JS Gli applicativi AngularJS seguono un particolare ciclo di vita in due fasi.

I vari moduli dell'applicazione vengono registrati nell'applicativo e ad essi vengono collegate le varie entità quali constants, services, components, filters eccetera.

Tutti questi elementi potranno essere istanziati solo in una seconda fase: grazie a questa divisione possiamo non preoccuparci dell'ordine con cui compaiono nel sorgente.

Possiamo dunque impiegare una banale prassi di concatenamento di files per condensare centinaia di sorgenti JS in un unico file perfettamente funzionante.

Le viste dell'applicativo sono centinaia quindi esiste nuovamente il problema di come veicolarle sul client. Non basta che queste siano semplicemente presenti nel bundle ma occorre che mantengano la correlazione con il loro componente.

Una soluzione potrebbe essere quella di codificarle come stringhe hardcoded direttamente nel component, ma sarebbe un codebase impossibile da mantenere. L'alternativa è invece quella di codificare i templates su file html separati ed iniettarli come stringhe inline a *build time* grazie al tool `gulp-angular-embed-templates`.

Per supportare invece la generazione di un artefatto con a bordo comportamenti diversificati in base all'ambiente per cui è destinato è stato adottato un approccio particolare.

Le specifiche configurazioni da usare sui vari ambienti sono espresse in files in formato YAML. Durante la fase di build, viene selezionato il file appropriato e con il tool `gulp-ng-constant` viene generato *on-the-fly* un file di definizione di una costante Angular. Questo sorgente generato sinteticamente viene quindi incluso nella build alla pari di tutti gli altri sorgenti. A runtime l'applicazione inietta questa costante in modo che la logica possa essere guidata dai valori espressi in fase di compilazione e non di codifica!

```
1  gulp.task('app:make_js', function(){
2
3    //1 - Collecting sources
4    return gulp.src([
5      path.join(cfg.appSrcDirPath, '**', 'module.js'),
6      path.join(cfg.appSrcDirPath, '**', 'main.js'),
7      path.join(cfg.appSrcDirPath, '**', 'state.js'),
8      path.join(cfg.appSrcDirPath, '**', '*.js')
9    ])
10
11   //2 - Embedding templates as inline strings
12   .pipe(angularEmbedTemplates())
13
14   //3 - Annotate components to help DI
15   .pipe(ngAnnotate())
16
17   //4 - Condensate multiple sources
18   .pipe(concat('app.js'))
19
20   //5 - Minify
21   .pipe(uglify({
22     mangle: false
23   }))
24   .pipe(rename({
25     suffix: '.min'
26   }))
27
28   //6 - Output file
29   .pipe(gulp.dest(cfg.jsOutputDirPath));
30 });
```

Listing 3.2: Stralcio della procedura di compilazione del JS dell'app.

Creazione dell'index.html Il file principale servito dal server al client cambia in base all'ambiente in cui è rilasciato.

La versione per l'ambiente di sviluppo include anche dei plugin del router che permettono di vedere il grafo degli stati *on-screen*.

La versione di produzione invece contiene i codici di integrazione con le piattaforme di tracking analitico di Google e Facebook, che non possono essere incluse nelle altre release altrimenti durante lo sviluppo si creerebbe un flusso dati che andrebbe a falsare le statistiche.

L'altra differenza sta nell'uso di librerie minificate o no. In produzione sono preferibili perché permettono di scambiare meno dati con il server ma durante lo sviluppo è più difficile lavorare con files di questo tipo.

Nonostante queste differenze, la struttura del documento html rimane sostanzialmente la stessa, quindi piuttosto che avere N diversi files (da mantenere allineati ad ogni modifica) si è scelto di usare un unico file *templatizzato*.

Il tool `gulp-nunjucks-render` permette di definire delle regioni nel documento e di sostituirne il contenuto in base a criteri a *build time*.

```
1  gulp.task('app:make_index', function(){
2
3    //Preliminary setup
4    [...]
5
6    //1 - Running template engine
7    .pipe(nunjucksRender({
8      data: data,
9
10     path: [   //where to look for partials
11       path.join(cfg.webIndexDirPath, 'partials', buildType),
12       path.join(cfg.webIndexDirPath, 'partials', 'common')
13     ]
14   }))
15
16   //2 - Output
17   .pipe(rename("index.html"))
18   .pipe(gulp.dest(cfg.outputDirPath));
19 });
```

Listing 3.3: Stralcio della procedura di compilazione dell'HTML dell'app.

3.5.2 Processo di build per la componente platform del sistema

La fase di build della componente lato server presenta molte meno difficoltà da gestire.

L'ambiente di esecuzione è più evoluto di quello del JavaScript e non c'è la rete di mezzo tra l'interprete e le risorse da eseguire.

Sostanzialmente i sorgenti PHP sono pronti da eseguire, quindi tra la fase di sviluppo e quella di rilascio non è necessaria nessuna fase di build.

Noi l'abbiamo voluta introdurre per permettere di commutare tra diversi set di configurazioni senza portare nel codice queste logiche legate all'ambiente.

Sostanzialmente l'applicazione è composta da un corredo di sorgenti fisso da includere sempre nella release e da un corredo variabile organizzato per directory relative all'ambiente.

La fase di build consiste nel copiare i files specifici per l'ambiente a cui è destinato il rilascio ed aggiungerli al corredo di files fisso. Per questo semplice compito sarebbe risultato troppo oneroso provisionare un ambiente Node.js e scomodare il tool Gulp.

Come verrà descritto nei capitoli successivi per la fase di deployment era già stato scelto il tool `Deployer`. Questo è scritto in PHP e non richiede interpreti aggiuntivi, quindi abbiamo deciso di implementare questa procedura di copia di files come un *task deployer*.

Tra questi files ne viene generato uno che permette di parametrizzare l'applicazione con il suo **numero di build** (questo compare nel backend e permette agli operatori di segnalarci i bug in maniera legata alle versioni).

```
1  function buildFor($env){
2
3    //Copy env files into source tree
4    run("cp -r ./environments/{$env}/* ./");
5
6    //Create 'buildNum.php' that is included via software
7    $buildNum = input()->getOption('buildNum');
8    $buildNumFileContent = "<?php return \"{$buildNum}\"; ?>";
9    run("echo '$buildNumFileContent' > ./protected/config/buildNum.php");
10 }
11
12 task('build:local', function(){
13   buildFor('local');
14 })->once();
15
16 task('build:staging', function(){
17   buildFor('staging');
18 })->once();
19
20 task('build:production', function(){
21   buildFor('production');
22 })->once();
```

Listing 3.4: Script di build scritto come task deployer.

3.6 Continuous Delivery

Per *Continuous Delivery* si intende il processo che permette di avere un flusso di sviluppo che possa scorrere in maniera stabile creando sempre una versione rilasciabile del sistema.

In letteratura esistono diversi approcci al problema [17], pertanto il *dove*, *quando* e persino *se* devono essere svolte certe parti del processo dipendono dalla situazione particolare ma il concetto chiave è quello di **mantenere il codice in uno stato che lo renda sempre rilasciabile**.

Facendo questa importantissima assunzione è possibile prevedere di **automatizzare tutto il processo a valle dello sviluppo** in modo che questo possa avvenire ripetutamente in maniera rapida ed affidabile senza intervento umano.

Questo approccio ingegneristico è **perfetto per le soluzioni come la nostra di tipo *Software as a Service***[10] perché la flessibilità che introduce permette di soddisfare più rapidamente e con maggiore qualità le richieste dei clienti finali.

Per le aziende l'introduzione di queste cose richiede un grande sforzo (soprattutto la prima volta) ma indagini di mercato[11]. confermano che questo trend di adozione è in aumento. L'altro dato che poi emerge è molte compagnie ritengono ancora il rilascio in produzione una attività strategica, da avviare a mano.

Anche noi siamo stati di questa idea ed abbiamo impostato il processo in modo che fosse possibile vedere il sistema finito prima di renderlo pubblico. Questo ha richiesto la predisposizione di un altro ambiente detto *staging* ed un setup di pipelines di rilascio maggiormente strutturato.

In particolare si è scelto che il **rilascio sull'ambiente di staging avvenisse automaticamente ed a partire da qualsiasi push su qualsiasi branch**.

Se ad esempio lo sviluppatore A esegue una push sulla sua feature branch avviene un rilascio. Se immediatamente dopo lo sviluppatore B esegue una push sulla sua (differente) feature branch avviene nuovamente un rilascio. In pratica sullo staging è visibile in anteprima la versione che è stata cronologicamente depositata per ultima sul repository remoto.

Per quanto questa scelta possa sembrare discutibile, ha senso. Ogni sviluppatore deve premurarsi di allineare il feature branch su cui sta lavorando con le novità che sono giunte nel frattempo sul master. Questo fa sì che in prossimità del merge della feature la storia del progetto (che si è evoluta parallelamente) sia stata in qualche modo già serializzata in una forma più lineare. Di fatto l'*head* di ogni feature branch rappresenta una futura release pubblica perché contiene già al suo interno la storia pregressa del master e consiste in una stratificazione di codice su di essa.

Decidere di rilasciare automaticamente (anche) le feature branch sullo staging fà sì che si possa vedere in anteprima il comportamento della prossima release del sistema, prima ancora di aver fuso il codice sul master. Questa scelta è saggia perché sul master non devono essere mai introdotte versioni instabili. Così facendo è possibile inserire una fase di revisione prima del merge, evitando così di considerare concluso lo sviluppo di feature che in realtà non lo era.

[10] *What is SaaS* https://azure.microsoft.com/it-it/overview/what-is-saas/

[11] *CD adoption survey report* https://www.perforce.com/pdf/continuous-delivery-report.pdf

Per quanto riguarda invece l'ambiente di produzione il funzionamento è diverso. Tutti i branch diversi dal master contengono per disciplinare aziendale versioni work in progress e non devono mai diventare di dominio pubblico.

Pertanto la pipeline è stata programmata in modo che il **rilascio sull'ambiente di produzione avvenisse manualmente ed a partire da una push sul solo branch master**.

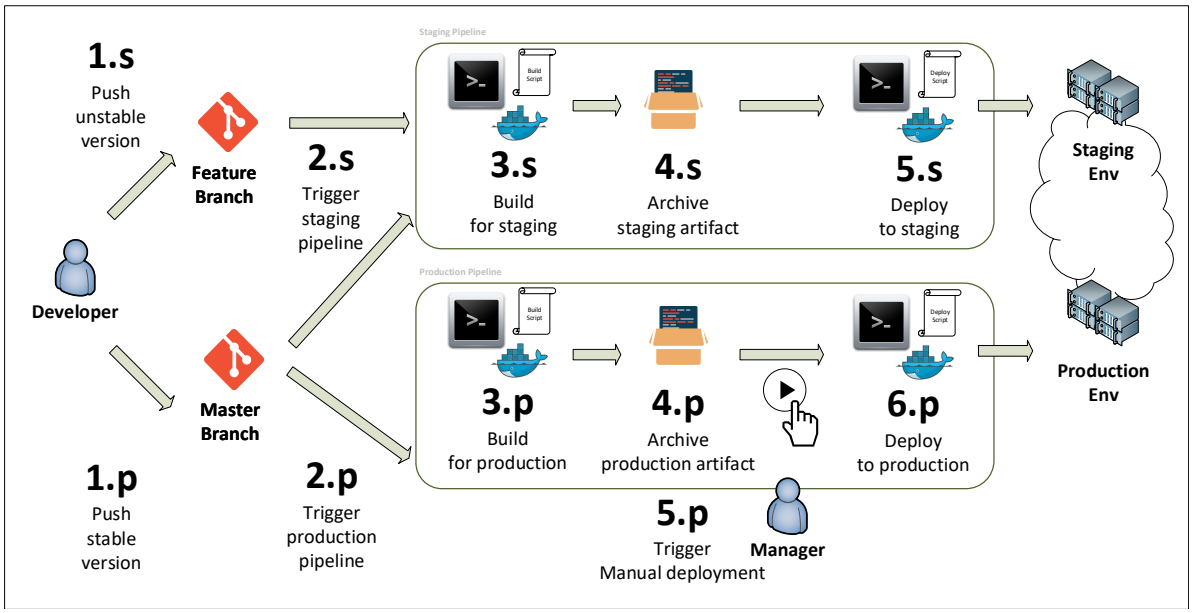

Figura 3.2: Le pipelines di staging e produzione.

L'iter di sviluppo del rilascio di una nuova versione è il seguente:

1. Quando il codice su una feature branch locale è maturo (poiché già testato in ambiente Vagrant) viene pushato sul remoto in modo da generare una *merge request*.
2. La pipeline di staging viene avviata producendo una build idonea per quell'ambiente ed effettuando il rilascio.
3. Sullo staging è possibile vedere in preview il comportamento del sistema `stable+feature`.
4. Se la revisione è positiva si può autorizzare il merge del feature branch sul master.
5. La pipeline di production viene avviata confezionato il package 'full optional' idoneo per lo scopo ed archiviandolo.
6. Il rilascio viene attivato a mano ed il package viene installato sull'ambiente di produzione.
7. Di fatto è appena stata pubblicata una nuova release. Se per qualche motivo dovesse emergere una criticità tale da giustificarne il ritiro, è possibile farlo rapidamente. Tutti gli artifatti archiviati al punto 5 possono essere usati come input per una nuova fase di deployment, quindi basta recuperare il penultimo e ripetere l'operazione.

Tutte queste informazioni (ossia il fatto che esistano gli `envs` staging e production e gli `stages` build e deploy e le regole di attivazione) sono espresse attraverso il file `.gitlab-ci.yml`.

```
stages:
  - build
  - deploy

 [...]

build:production:    #Job for build stage on production env

  stage: build
  environment:
    name: production

  image: node:9    #https://hub.docker.com/_/node/

  script: [...]

  artifacts: #What to consider output of this job
    paths:
      - public_html/

deploy:production: #Job for deploy stage on production env

  stage: deploy
  environment:
    name: production

  when: manual    #Trigger manually
  only:         #Run pipeline only on push to master
  - master

  dependencies:    #Using artifacts of 'build:production'
    - build:production

  image: ubuntu:xenial   #https://hub.docker.com/_/ubuntu/

  variables:
    GIT_STRATEGY: none #No sources required during deploy

  script: [...]
```

Listing 3.5: File `gitlab-ci.yml` di esempio.

Per ogni `job` viene indicato lo stage e l'ambiente in cui deve essere eseguito.

Il nodo `script` indica le istruzioni da eseguire per quel job. Tipicamente si referenzia qui uno script di shell o si elencano direttamente i comandi (il tutto viene eseguito in un ambiente *headless* a linea di comando).

Ogni job esprime anche l'ambiente in cui devono essere eseguiti i comandi attraverso il nodo `image`. Il runner di GitLab si collega al repository pubblico di containers Docker[12] e scarica quello indicato.
Dal momento che le nostre procedure di build erano già state implementate per rendere gli applicativi eseguibili dentro le macchine Vagrant, bastava selezionare dei container Docker che avessero un corredo di tool sufficiente per eseguire tali procedure.

Attraverso la coppia di nodi `artifact` e `dependencies` è possibile indicare l'output di un job e quanto richiesto in input per un altro job. Di fatto in questo modo è possibile *passare* dei files tra un job e l'altro. Inoltre questi file vengono archiviati su GitLab in modo che sia possibile scaricare dall'interfaccia o ripetere di deploy di qualsiasi build del sistema.

I nodi `when` e `only` permettono di filtrare quali sono i branch che determinano l'esecuzione della pipeline ed il tipo di attivazione. Quando viene parametrizzato con *manual* l'interfaccia grafica di GitLab provvede a mostrare un bottone *play* apposito per attivare l'esecuzione.

Il sistema esegue gli script in un contesto in cui sono definite delle *variabili di ambiente*. Alcune sono pre definite dal sistema[13] altre possono essere definite dall'utente attraverso l'interfaccia grafica di GitLab.
Queste variabili possono essere usate come input (ad esempio durante l'esecuzione dello script è possibile leggere la variabile `CI_BUILD_REF` per conoscere il numero di commit associato alla versione che si sta processando).
Certe variabili possono essere ridefinite per cambiare dei comportamenti di default.

[12] *Docker Hub* https://hub.docker.com/search?q=&type=image

[13] *GitLab Environment Variables* https://docs.gitlab.com/ee/ci/variables/#predefined-environment-variables

3.6.1 Pipeline 'app'

3.6.1.1 Job di 'build'

La procedura di build è già implementata sotto forma di task Gulp, pertanto per essere eseguita risultano necessari:

Ambiente di esecuzione Node.js Ottenibile scegliendo l'immagine Docker `node:9` che il sistema reperirà automaticamente all'indirizzo `https://hub.docker.com/_/node/`

Tool Gulp e relative dipendenze Tutti i tools e librerie dell'ecosistema Node sono reperibili attraverso il package manager `npm` (già installato nell'immagine scelta).
Avendo avuto cura di compilare il file `package.json` del progetto è sufficiente eseguire `npm install` per provisionare il container virtualizzato di tutto il necessario per eseguire il task.

```
1  #Preparing build environment
2  npm install -g bower gulp-cli
3
4  #Fetching toolset
5  npm install
6
7  #Fetching app dependancies
8  bower install --allow-root
9
10 #Performing build invoking the Gulp task named 'build'
11 gulp build  --build_type="release"
12            --build_env="$CI_ENVIRONMENT_NAME"
13            --build_num="$CI_BUILD_REF"
```

Listing 3.6: Script di build 'app' referenziato da `gitlab-ci.yml`.

3.6.1.2 Job di 'deploy'

La procedura di deploy è estremamente minimale in quanto per effettuare il rilascio dell'app è sufficiente depositare via `FTP` i file elaborati nello step precedente (disponibili nella directory `public_html`). I requisiti sono dunque:

Distribuzione Linux qualunque Abbiamo scelto l'immagine `ubuntu:xenial` che il sistema reperirà automaticamente all'indirizzo `https://hub.docker.com/_/ubuntu/`

Client FTP Abbiamo scelto il client `lftp` che è disponibile nei repository ufficiali e può essere installato tramite il comando `apt-get install lftp`. Invece che codificare le credenziali di accesso all'host remoto (cosa che avrebbe sollevato un rischio di sicurezza) sono state inserite tramite l'interfaccia grafica di GitLab in modo che fossero disponibili durante l'esecuzione come variabili d'ambiente.

```
1 #Preparing deploy environment
2 apt-get update
3 apt-get install -y lftp
4
5 #Mirror local directory 'public_html' to remote doc root
6 lftp -c "open -u $PRODUCTION_FTP_USER,$PRODUCTION_FTP_PASSWORD
     $PRODUCTION_FTP_HOST set ssl:verify-certificate no; mirror -R
     public_html/ /app.il730.online"
```

Listing 3.7: Script di deploy 'app' referenziato da `gitlab-ci.yml`.

3.6.2 Pipeline 'platform'

La procedura di deployment per il sistema platform è più complessa di quella dell'app perché in questo caso non basta depositare dei file sull'host destinatario ma bisogna anche eseguire dei comandi a bordo di questo ambiente.
Questo discorso fà particolare riferimento alla necessità di applicare le **migrations** per allineare la struttura del database a quella del software.

Vista la complessità di questa attività abbiamo scelto di usare il **tool specifico** `Deployer` in quanto prevede già un trattamento (nel suo gergo detto *recipe*) idoneo per progetti realizzati con il framework `Yii`.

Questo è pensato per essere eseguibile anche come tool standalone al di fuori di in una pipeline di delivery, pertanto **incorpora già tutte le funzionalità necessarie per effettuare un rilascio partendo dai sorgenti**.

Per questo motivo lo stage di build nel senso stretto della pipeline di GitLab non esiste perché tutte le attività sono centralizzate in un unico stage cumulativo implementato delegando l'intera attività al suddetto strumento.

3.6.2.1 Job di 'build & deploy'

Nei capitoli precedenti è stata introdotta una nozione di build per il sistema platform implementata sotto forma di task Deployer. Essendo anche l'intera procedura di deployment implementata con questo tool, **questo è tutto ciò di cui si necessita** per eseguire la pipeline platform.

Come già anticipato altrove, c'è la necessità di eseguire comandi sull'host remoto e questo può essere fatto con una sessione autenticata SSH.
L'uso di credenziali del tipo `username+password` è una prassi scoraggiata: è invece preferibile usare un'*identità RSA*.

1. Il client genera la sua identità *una-tantum* invocando il comando:
   `ssh-keygen -t rsa -C \deployer" -b 4096`

   L'identità appena creata servirà per impersonare l'attore *deployer* nei confronti di host remoti. Questa consiste in una coppia di files: uno rappresentante la chiave privata e l'altro la sua controparte pubblica.

2. La chiave pubblica appena generata deve essere depositata su tutti i servers ai quali ci si vuole connettere in maniera *passwordless*. Anche questa attività è da eseguirsi una tantum usando il mezzo di trasferimento più comodo per l'occasione (es. `SCP` con utente root o consegna manuale dei file all'amministratore del server).

3. A questo punto il client può avviare una connessione SSH senza password verso uno dei server precedentemente configurati. Il client critta un messaggio con la sua chiave privata, impersonando di fatto l'identità dell'*attore deployer*. Il server riuscirà a riconoscere l'autore del messaggio tra quelli autorizzati per il fatto di riuscire ad impiegare con successo la corrispondente chiave pubblica che ha a bordo.

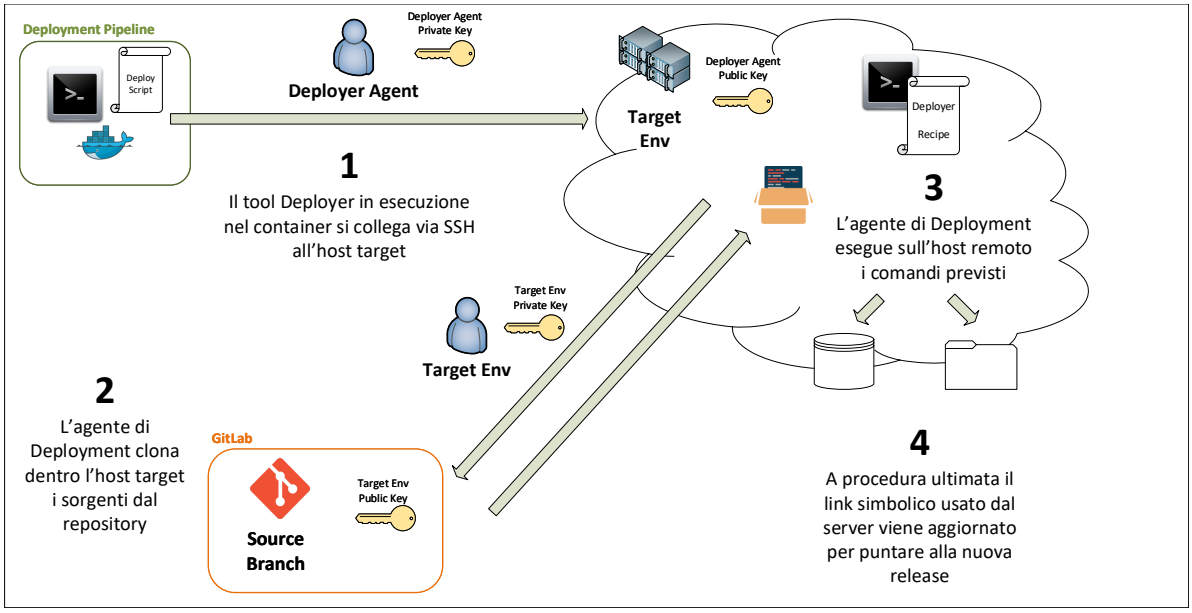

Figura 3.3: Processo di deployment sul sistema platform con l'uso del tool Deployer.

1. Il runner GitLab processa lo script di definizione della pipeline di deployment ed accende un container idoneo ad eseguire il tool `Deployer`. Il tool è scritto in PHP quindi l'immagine Docker scelta è la `php:5.6-cli` reperibile all'indirizzo `https://hub.docker.com/_/php/`

   Questo tool effettua come prima attività una connessione passwordless SSH al server destinatario. Affinché la connessione possa avvenire con successo occorre che venga usata una identità prescelta (che chiameremo dell'*agente deployer*) e che sia stata precedentemente installata sul server target (immettendo la chiave pubblica tra quelle autorizzate).

   Lato client occorre avviare la connessione usando la chiave privata. Và considerato che questo processo viene eseguito in un container Docker creato e distrutto ogni volta *ex-novo*, quindi occorre ad ogni avvio della pipeline provvedere installare questa chiave affinché possa essere usata.

   Per non sollevare questioni di sicurezza, la chiave privata dell'agente deployer è stata precedentemente inserita come variabile segreta dentro GitLab. In questo modo è disponibile come variabile d'ambiente durante l'esecuzione della pipeline e può essere impiegata senza che compaia in nessun luogo pubblico.

   Alla fine di questa fase il tool `Deployer` (in esecuzione sul container) è in grado di eseguire comandi sull'host target.

2. Ora il tool crea una nuova directory di appoggio ed avvia una operazione di `git clone` per recuperare i sorgenti.

   Questa volta la connessione passwordless avviene tra l'host target ed il repository. GitLab è progettato per questo tipo di interazioni e prevede un'apposita interfaccia grafica per

immettere le così dette *Deploy Keys*, ossia le chiavi pubbliche degli host che sono abilitati a clonare i sorgenti.

Alla fine di questa fase nella directory di appoggio sul server target sarà presente una copia dei sorgenti.

3. A questo punto il tool Deployer può eseguire i task personalizzati (come il nostro di *build* menzionato nei capitoli precedenti) e tutte le attività previste dalla *recipe*.

   Il nostro progetto è costruito con il framework Yii e la sua apposita recipe prevede già l'**applicazione delle migrations**. Questo avviene nel contesto di una *transaction SQL*, così che nel caso dovessero emergere problemi tutte le modifiche fatte possono essere *rollbackate* per lasciare il database in uno stato consistente.

   Un'altra attività già prevista dal tool è quella del **recupero delle dipendenze** attraverso il package manager `Composer`.

4. Se tutte le procedure di preparazione della build sono andate a buon fine si può committare la transaction SQL ed aggiornare il puntamento del virtual host di Apache verso la nuova directory appena preparata.

   In particolare segnaliamo che il virtual host è configurato per referenziare un path che a livello di sistema corrisponde ad un link simbolico. In questo modo risulta veramente semplice passare da una versione ad un'altra.

```
1  #Disable host key checking
2  mkdir -p ~/.ssh
3  echo -e "Host *\n\tStrictHostKeyChecking no\n\n" > ~/.ssh/config
4
5  #Prepare for impersonating 'deployer' identity
6  echo "$DEPLOYER_SSH_PRIVATE_KEY" > ~/.ssh/id_rsa
7  chmod 400 ~/.ssh/id_rsa
8
9  #Installing SSH client
10 apt-get update
11 apt-get install -y ssh
12
13 #Installing Deployer 4.x
14 curl -LO https://deployer.org/releases/v4.3.1/deployer.phar
15 mv deployer.phar /usr/local/bin/dep
16 chmod +x /usr/local/bin/dep
17
18 #Deploying revision $CI_BUILD_REF using Deployer
19 dep deploy production
20           -vvv
21           --revision="$CI_BUILD_REF"
22           --buildNum="$CI_BUILD_REF"
```

Listing 3.8: Script di deploy 'platform' referenziato da `gitlab-ci.yml`.

4 Conclusioni

4.1 Validazione

L'azienda in cui lavoro al momento della presentazione di questa tesi adotta un **percorso di qualità certificato**, pertanto già da prima di questo progetto eravamo tutti abituati a gestire il lavoro in *fasi* ed a produrre le *evidenze* delle attività svolte.

Aziendalmente usiamo un software ti tipo CRM per storicizzare le richieste dei clienti ed il lavoro fatto sui contratti.

Ogni volta che commercialmente viene conclusa una vendita il responsabile di reparto provvede ad inserire il contratto nel sistema in modo che ogni addetto possa sapere *cosa* è stato venduto e qual'è il servizio da erogare. Inoltre provvede a quantificare il budget venduto in un *monte ore* da spendere in attività di reparto, spacchettandolo in vari *service contracts*.

Ogni volta che un addetto svolge una attività su un progetto deve aprire un *ticket* di lavoro collegato ad una di queste entitità ed annotare al suo interno la **motiviazione** dell'attività svolta, la **soluzione** sviluppata e le **ore impiegate**.

In questo modo è possibile **monitorare in tempo reale il consumo del monte ore** e rapportarlo allo stadio di avanzamento dei lavori.

La quantità di ticket presenti sui service contracts di tipo *service* danno un'indicazione della complessità del progetto. Avere tanti ticket nello storico significa che sono state richieste tante micro attività. Il loro totale ore complessivo rapportato al monte ore previsto indica **quanto margine economico c'è stato sul progetto**. Questa informazione è importantissima perché permette alla direzione di fare delle valutazioni sul fatto che i progetti vengano venduti con i giusti budget. Allo stesso modo permette di attivare in anticipo delle contromisure operative quando si intuisce che il progetto potrebbe sforare il budget.

Nel caso de IL 730 ONLINE nonostante l'enorme mole di lavoro che ha richiesto **siamo riusciti a rientrare nel budget** con grande soddisfazione della direzione. Questo dimostra che siamo stati in grado di realizzare il lavoro nei tempi previsti e di aver fatto un computo preliminare delle attività congruo. Il mio contributo è stato importante anche in questa fase di valutazione iniziale sulla fattibilità e sull'economicità della cosa.

La proporzione di ticket di tipo *support* rispetto al tempo, dà un'indicazione di quanto lavoro è servito dopo la fine del progetto. Anche per questo aspetto possiamo ritenerci soddisfatti in quanto **le richieste di assistenza sono state poche e non sono mai emersi bug critici**.

Per quanto il concetto di *qualità* sia difficile da quantificare credo che questi siano dati oggettivi che denotano un livello indiscutibile di bontà del prodotto rilasciato.

Il software GitLab introdotto per la prima volta nell'occasione di questo progetto ha aggiunto un ulteriore livello di dettaglio a questa capacità di monitorare il nostro operato. Le sue funzionalità di reportistica permettono infatti di vedere in maniera grafica il trend di contribuzione al progetto ed il flusso di segnalazioni in entrata ed i rilasci in uscita.

Questi due strumenti (CRM & GitLab) uniti alla mail aziendale ci permettono di risalire in pochissimo tempo a qual'era stato il motivo e l'esito di ogni micro attività svolta in qualsiasi momento temporale di un progetto.

Sono fortemente dell'idea che strumenti di questo tipo dovrebbero essere impiegati in qualsiasi attività di business perché non c'è tecnologia o competenza che può portare al successo senza un dovuto controllo e monitoraggio. Questa visione delle cose l'ho maturata in questi ultimi 5 anni di lavoro in HI-NET e posso affermare che è una delle cose più belle ed utili che ho imparato.

Ogni lunedì svolgiamo una riunione di reparto in cui si fà il punto della situazione, commentando le attività fatte ed anticipando le criticità.

Il disciplinare aziendale prevede inoltre dei **checkpoint di interazione con il cliente** nei quali lo si coinvolge in un processo di *validazione* atto a raccogliere una documentazione utile per dimostrare che in quella fase era soddisfatto del lavoro ed autorizzava a procedere a quella successiva.

Ogni anno l'ente che rilascia la certificazione di qualità esegue un esame in azienda per verificare che i progetti siano stati portati avanti nel rispetto dei dettami concordati. Anche da questo punto di vista siamo risultati pienamente conformi quindi possiamo andare fieri del nostro operato.

4.2 Benefici Misurabili

Nella trattazione che seguirà verranno ripresi i punti salienti del workflow discusso nei capitoli precedenti evidenziando i vantaggi diretti ed indiretti che hanno portato.

4.2.1 Benefici sull'operato aziendale

Dati dall'uso di postazioni virtualizzate con Vagrant

- Grazie alla virtualizzazione si può lavorare da Windows usando i software di produttività più comuni ma eseguendo il software con un supporto nativo delle feature che tipicamente mancherebbero (user permission, symlink, struttura diversa del filesystem, path case sensitive).

  Dal momento che la VM viene costruita usando la stessa immagine dell'ambiente di produzione, si riescono a replicare sull'ambiente di sviluppo anche quei bug che inevitabilmente esistono a livello di sistema e le particolari configurazioni.

  Tutto questo permette di **scovare i bug con maggiore anticipo**.

- In fatto che la workstation di per sè non venga interessata dall'installazione di strumenti fà sì che per ogni progetto si possa disporre di uno specifico toolset **senza avere conflitti**.

- Il fatto che l'ambiente virtualizzato sia ricreabile **svincola lo sviluppatore da una specifica postazione di lavoro**. Inoltre permette di adottare durante lo sviluppo approcci distruttivi perché nel caso, il ricominciare da zero sarebbe una operazione senza costo.

- Il costo di avvio di questo approccio (ossia scrivere il vagrantfile e gli script di provisioning) **si ammortizza brevemente nel tempo** perché bene o male tutti i progetti hanno una configurazione alla base comune e si può dunque riciclare codice e soluzioni.

Dati dall'adozione della metodologia Feature Branch

- I conflitti di versione quando emergono vanno gestiti: non esiste alternativa. Il fatto di anticipare questa attività permette di **distribuire meglio il carico di lavoro** ed evitare di concentrare attività critiche in momenti a ridosso della consegna.

- Il fatto di avere sempre un branch con versioni stabili permette di non avere mai intralci nel fare rilasci velocemente. Questo permette di **essere percepiti dal committente come una azienda reattiva** ad esaudire le richieste.

- **É facile risalire alla storia del progetto** in quanto nel branch mainline tutte le versioni derivate da un merge corrispondono all'introduzione di una feature stabile e conseguentemente ad una *release* del sistema.

Dati dall'adozione della Continuous Delivery

- Essendo l'attività di rilascio eseguita dall'automa, non serve concentrazione per farla quindi **il team in generale subisce meno stress**.

  Grazie a questo fatto è possibile eseguire rilasci anche in momenti di grande pressione (ad esempio durante consegne multiple) e questo si traduce in **maggiore produttività**.

- Il tempo necessario per il rilascio viene ridotto di un ordine di grandezza (dalle ore ai minuti) questo vuol dire **riduzione dei costi**.

- L'attività di rilascio viene svolta da automi che operano da una posizione sulla rete nota. Forti di questa informazione è possibile limitare la connettività agli ambienti finali lasciando aperti solo questi canali.

  Analogamente non ci sono chiavi di accesso condivise tra individui (ad esempio anche lo stagista di turno potrebbe effettuare un rilascio senza venire in possesso di nessuna informazione sensibile).

  L'infrastruttura di rete che ne deriva è quindi **più sicura contro malintenzionati**.

- Lo sviluppo al progetto **non è vincolato ad uno specifico IDE**. In situazioni di emergenza (ad esempio lavorando da remoto o in viaggio) è possibile apportare una correzione al codice tramite l'editor web based di GitLab e fare un rilascio disponendo solo di una connessione Internet!

- In caso di errori durante la procedura di rilascio nulla viene intaccato, mentre è possibile in ogni caso fare il rollback ad una versione precedente se lo si vuole.

  Questo contribuisce all'obiettivo di avere **software resiliente** [18].

- Chiunque è in grado di avviare un processo di rilascio senza avere competenze specifiche: questo permette di **impiegare meglio il personale**.

- Il fatto di avere rilasci *effortless* permette di avere una spirale del software a raggio più corto, quindi ottenere il progresso del progetto attraverso l'introduzione di micro passi, sostenendo **minori rischi**.

4.2.2 Benefici sul prodotto finito

Nelle prime fasi del progetto i rilasci venivano effettuati a mano ed ancora non era in piedi un sistema automatizzato per produrre delle build multi-ambiente. Anche questa attività veniva fatta manualmente.

Il livello di pressione in questi momenti era notevole e nonostante la grande concentrazione che veniva impiegata qualche volta capitava di commettere degli errori mettendo online una release corretta dal punto di vista del codice ma non funzionante.

Inoltre l'intero processo durava un tempo non trascurabile, durante il quale il sistema andava offline.

Da quando abbiamo implementato gli script di automazione ed in seguito introdotto le pipelines su GitLab abbiamo avuto **zero down-time durante i rilasci** e non è **mai più capitato di mettere online rilasci *spaccati***. Questi sono dati oggettivi inopinabili.

Un'altro fatto degno di nota è quello relativo alla capacità di erogare un servizio efficiente di *support*. Certe volte è capitato che giungessero segnalazioni di bug fatte da contribuenti che erano rimasti bloccati perché il sistema non gli permetteva di procedere. Quelle volte in cui è stato possibile affrontare il problema immediatamente siamo stati in grado di **risolvere il bug ed effettuare il rilascio con il cliente in attesa dall'altra parte**!

Per dare alcuni numeri all'esempio appena fatto:

Tempo di raccolta della segnalazione del bug Il sistema è ben fatto e quando accade un problema viene sempre mostrata una idonea messaggistica.
Questo ci permette di risalire quasi sempre allo scenario erroneo in **qualche minuto**.

Tempo di creazione in locale del problema Grazie all'ambiente virtualizzato possiamo scaricare un *dump* dei dati di produzione ed impersonare quell'utente per riprodurre i passi che rendevano il bug manifesto.
Anche questa attività richiede **qualche minuto**.

Correzione del problema Il tempo varia ovviamente dalla complessità del problema emerso. La storia del progetto insegna che sostanzialmente non sono capitati problemi che hanno richiesto giorni per essere affrontati. In molti casi il bug era così semplice che poteva essere affrontato in **circa 10 minuti**.

Rilascio Grazie alla possibilità di vedere in preview la correzione ed eseguire localmente i test che la interessano, abbiamo una notevole confidenza nel codice che rilasciamo. Le pipelines automatiche impiegano mediamente **3 minuti** per prelevare i sorgenti ed effettuare un deployment completo.

Sostanzialmente **in un caso ottimale, dal momento della segnalazione del bug al rilascio della correzione trascorrono circa 20 minuti**.

Và poi sottolineato che grazie alla natura SaaS della soluzione, la correzione non è attiva per il solo utente che l'aveva segnalata ma per l'intero bacino utenti.

Al momento della presentazione di questa tesi non è scontato vedere in soluzioni commerciali una capacità di evoluzione del genere senza richiedere al contempo alcuna attività da parte dell'utente.

4.3 Potenziali Criticità

Dipendenza dai package managers Il fatto che tutte le procedure vengano eseguite ogni volta in ambienti virtualizzati *vergini* creati per l'occasione, fà sì che sia indispensabile avvalersi di package managers per recuperare la mole di software e tool necessari per eseguire il processo al loro interno.

Questo introduce una dipendenza importante: nel momento in cui l'autore di un tool decidesse di distribuirlo attraverso un nuovo canale occorrerebbe adeguare la procedura per integrarla anche con questo nuovo servizio.

Analogamente vale lo stesso discorso sull'intero sistema di package manager, che potrebbe prima o poi venire dismesso.

Questo caso è avvenuto realmente. Dopo aver avviato il progetto ed aver scelto `Bower` come package manager front-end, hanno iniziato a comparire dei warning[1] durante il suo utilizzo che invitavano ad adottare `Yarn` al suo posto per i nuovi progetti.

Attualmente non abbiamo preso alcuna contromisura ma ci aspettiamo che da un momento all'altro venga completamente chiuso: a quel punto dovremo per forza di cose re implementare la gestione dei *vendors* nella fase di build dell'app oppure recuperare l'ultimo snapshot delle dipendenze ed includerle nel progetto come risorse statiche.

La seconda alternativa, sebbene più semplice da operare invalida completamente il concetto di fondo dell'uso di un package manager, ossia la capacità di amministrare le dipendenze recuperandole da un repository senza effettuare download manuali e mantenendo il progetto aggiornato con gli ultimi rilasci di queste ultime.

I problemi nel trattamento delle librerie front-end possono essere affrontati in questo modo semplice, ma per gli altri tipi di dipendenze (quelle della toolchain Node e quelli dei sistemi Ubuntu usati nelle pipelines) non esiste alternativa.

L'unico modo di svincolarsi dai repository è quello di preconfezionare delle immagini dei sistemi che hanno già a bordo i tools in modo da non dover reperire nulla *on-the-fly*.

Questo approccio è quello che dà maggiore sicurezza a fronte di stravolgimenti futuri in quanto l'immagine congela lo stato delle cose al momento della sua creazione. Il prezzo da pagare però è quello di un maggiore sforzo iniziale. L'immagine deve essere creata (manualmente) e pubblicata in un posto dove sia accessibile da parte dei vari suoi client, quali il tool vagrant che preleva la box di base ed il runner GitLab che preleva l'immagine Docker da usare nella pipeline. A scapito di una maggiore sicurezza oltre a questo costo *infrastrutturale* emerge un costo latente operativo. Ogni volta che si decide di adeguare una procedura utilizzando un nuovo tool, tutte le immagini pubblicate devono essere aggiornate.

Probabilmente non esiste una soluzione perfetta: si tratta di un ennesimo compromesso da ottimizzare progetto per progetto.

[1] *How to migrate away from Bower?* https://bower.io/blog/2017/how-to-migrate-away-from-bower/

Difficoltà nel preview con devices mobili Il fatto che il progetto sia eseguito dentro ad un ambiente virtualizzato desktop rende difficile il suo accesso da parte di un dispositivo mobile fisico.

Smartphones e tablets hanno come unico canale di connessione quello WiFi e non permettono di amministrare in maniera avanzata i parametri di rete: l'unico modo per configurarli è quello di disporre di una rete WiFi speciale che via DHCP proponga un *server DNS truccato* che risolva il dominio del sistema verso l'IP della macchina Vagrant. Dal momento che ogni sviluppatore ha una sua VM sulla sua workstation occorrerebbero N reti WiFi così configurate.

L'alternativa è quella di usare i simulatori software di device. Questo approccio non lo troviamo di valore per i nostri scopi in quanto non permette di verificare la vera esperienza d'uso e le performance con devices di fascia bassa.

Il compromesso che abbiamo adottato è stato quello di implementare la WiFi di sviluppo nei confronti dell'host di staging. In questo modo è possibile provare su device fisico il comportamento del sistema prima di renderlo pubblico ma il prezzo da pagare per avere questa preview è quello di fare un deployment provvisorio.

4.4 Sviluppi futuri

Il sistema così strutturato potrebbe potenzialmente essere sviluppato orizzontalmente aggiungendo la fornitura *online* di qualunque servizio erogato dal CAF (es. modello ISEE, contratti di affitto eccetera). Il principio di base è sempre il medesimo: una volta confermata l'identità digitale si può procedere ad inviare documenti e comunicazioni fino a ricevere un documento scaricabile previa esecuzione di un pagamento.

Uno sviluppo verticale basato su quanto già fatto potrebbe consistere nell'implementazione di politiche di gruppo per utenti affiliati.

Al momento della presentazione di questa tesi, in Italia sta emergendo il mercato dei servizi nel contesto del *welfare aziendale*. Sostanzialmente le aziende acquistano in blocco una mole di servizi da certe società e li destinano ai loro dipendenti: in questo modo il vantaggio esiste per tutti. L'azienda che acquista i servizi sostiene dei costi attraverso i quali può avere dei benefici fiscali, il dipendente vede scontato dalla propria paga un costo minore di quello che avrebbe dovuto sostenere acquistando il servizio da solo e la società erogante riesce a concludere in un'unica operazione una vendita massiva.

Dal momento che la prassi della dichiarazione dei redditi tocca tutti i lavoratori dipendenti può essere questo un mercato interessante. Dal punto di vista tecnico il sistema dovrà dotarsi di strumenti di backoffice per immettere quali sono i soggetti beneficiari dell'affiliazione ed il sistema frontend dovrà reagire facendogli saltare certe fasi (quali ad esempio il pagamento).

Un possibile sviluppo sul fronte tecnico potrebbe aver senso per dare un supporto ancora migliore agli utenti mobile. Convertendo la webapp in app nativa potremmo fare un uso più a basso livello della fotocamera e potremmo introdurre ad esempio algoritmi di correzione geometrica per rimuovere la prospettiva dagli scatti fotografici dei documenti (come già fanno certe app sul mercato[2]).

Evolvendo invece la SPA in PWA potremmo beneficiare della tecnologia del *service worker* per eseguire attività in background che tipicamente non possono essere fatte in una app tradizionale per via della natura single-threaded dell'ambiente di esecuzione JavaScript. Sui dispositivi mobile il service worker continua a viviere anche dopo la chiusura della finestra del browser, permettendo ad esempio di continuare ad uploadare i documenti anche dopo l'abbandono da parte dell'utente.

[2] *Microsoft Office Lens* https://play.google.com/store/apps/details?id=com.microsoft.office.officelens&hl=it

Bibliografia

[1] Consulta Nazionale dei CAF. Oltre 17.000.000 gli italiani che si sono rivolti ai caf per la dichiarazione dei redditi (mod. 730) e 2.700.000 le dichiarazioni online. https://www.consultacaf.it/oltre-17000000-gli-italiani-che-si-sono-rivolti-ai-caf-per-la-dichiarazione-dei-redditi-mod-730-e-2700000-le-dichiarazioni-online/, 2018. [Online; accessed 27-Jan-2019].

[2] CAF ACLI. Chi Siamo. http://www.caf.acli.it/chi-siamo.html, 2019. [Online; accessed 27-Jan-2019].

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[4] Martin Fowler. Feature branch. https://martinfowler.com/bliki/FeatureBranch.html. [Online; accessed 3-Feb-2019].

[5] Atlassian. Git feature branch workflow. https://it.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow. [Online; accessed 3-Feb-2019].

[6] M. DiPierro. The rise of javascript. https://doi.ieeecomputersociety.org/10.1109/MCSE.2018.011111120, January/February 2018.

[7] Hazem Saleh. *JavaScript mobile application development : create neat cross-platform mobile apps using Apache Cordova and jQuery Mobile*. Packt Pub, Birmingham, U.K, 2014.

[8] Jeremy Wilken. *Ionic in action : hybrid mobile apps with Ionic and AngularJS*. Manning Publications, Shelter Island, New York, 2016.

[9] AirbnbEng. Isomorphic javascript: The future of web apps – airbnb engineering & data science – medium. https://medium.com/airbnb-engineering/isomorphic-javascript-the-future-of-web-apps-10882b7a2ebc, Nov 2013.

[10] Minko Gechev. *Switching to Angular 2 : build SEO-friendly, high-performance single-page applications with Angular 2*. Packt Publishing, Birmingham, UK, 2016.

[11] Adam Freeman. *Pro AngularJS*. Apress,Distributed to the Book trade worldwide by Springer, Berkeley, CA New York, NY, 2014.

[12] Continuous integration, delivery, and deployment with gitlab. https://about.gitlab.com/2016/08/05/continuous-integration-delivery-and-deployment-with-gitlab/, Aug 2016.

[13] How valuable is amazon's 1-click patent? it's worth billions. http://rejoiner.com/resources/amazon-1clickpatent/, May 2017.

[14] David Flanagan. *JavaScript : the definitive guide*. O'Reilly, Sebastopol, CA, 2006.

[15] Kristin Brennan. The rise of web technology. `https://www.sencha.com/blog/the-rise-of-web-technology/`, Jun 2015.

[16] Mark Safronov. *Web application development with Yii 2 and PHP : fast-track your web application development using the new generation Yii PHP framework*. Packt Pub, Birmingham, UK, 2014.

[17] Continuous Deployment For Practical People. `https://www.airpair.com/continuous-deployment/posts/continuous-deployment-for-practical-people`. [Online; accessed 23. Feb. 2019].

[18] MikeWasson. Progettazione di applicazioni resilienti per Azure. `https://docs.microsoft.com/it-it/azure/architecture/resiliency/index`, Dec 2018. [Online; accessed 24. Feb. 2019].