* La SOA définit les concepts qui permettent de mettre en place des SI articulés autour de services et communiquant de façon standardisée.
* Un contrat WSDL est un document XML qui explique les fonctionnalités d’un service, comment l’appeler, où le trouver et quels types de réponses il renvoie.
* UDDI est un serveur qui centralise tous les contrats des services.
* Les services doivent communiquer via SOAP : un protocol XML standard.
* En Java, on utilise des librairies comme JAX-WS pour générer les documents WSDL et les messages SOAP.

**8 commandements de la SOA**

Pour vous assurer que votre architecture SOA est fiable et exploiter au maximum son potentiel, voici 8 règles fondamentales à respecter.

### Un contrat standard vous appliquerez

Il est important que les documents WSDL soient standarisés au maximum afin de faciliter les réutilisabilités des services.

### Un couplage faible vous appliquerez

Vos services doivent êtres indépendants et faiblement couplés aux autres composants du SI. Ceci se concrétise en veillant à ce que tous les échanges de votre service avec le monde extérieur se fassent uniquement via SOAP.

### Abstraits vos services seront

Les contrats que vous publiez doivent indiquer le minimum vital nécessaire à l’invocation de votre service. En aucun cas vous ne devez publier des informations sur le fonctionnement interne du service.

### Des services réutilisables vous développerez

La réutilisabilité d’un service se mesure à son indépendance de la logique de son processus métier.

### Autonomes vos services seront

Quand vous développez un service web, faites en sorte qu’il soit le plus autonome possible, c'est-à-dire qu’il dispose de ses propres ressources, libraries, containers, etc.

### Sans état (stateless) vos services seront

Quand vous développez un service, il faut veiller qu'aucune requête ne dépende de la réponse d’une autre.

### Découvrabilité

Comme on l’a vu plus tôt dans le cours, votre service doit pouvoir être trouvé facilement par les autres services grâce à la publication de son contrat (WSDL) dans un annuaire de type UDDI.

### Composabilité

Votre service doit maintenant participer à la composition d’une application ou d’un SI plus large en proposant ses fonctions à différents composants. La réutilisabilité du service doit être exploitée au maximum afin d’optimiser le système final.

**Pour vous aider à découper vos web services** de manière à ce qu’ils soient compatibles avec une SOA, il faut respecter les 2 règles suivantes :

* Votre web service doit être IRA.
* Votre web service doit respecter les 8 commandements.

Qu’est-ce qu’un web services IRA ?

**I** : Interesting

**R** : Reusable

**A** : Atomic

***Interesting*** :

Les fonctionnalités de votre web service doivent être intéressantes non seulement pour servir la finalité du composant que vous développez mais potentiellement intéressantes pour l’ensemble des web services du système.

*Exemple :* Un service qui met à jour un champ dans une table de base de données dès qu’une commande est ajoutée est totalement inintéressant, car il s’agit d’un service qui traite du fonctionnement interne du processus de commande (par exemple dans la boutique en ligne). Aucun autre web service ne sera intéressé par la mise à jour d’un champ obscur dans une table aux tréfonds de la base de données d’un CMS !

***Reusable*** :

Votre web service doit être réutilisable dans d’autres contextes et scénarios. Il ne doit pas être lié à la logique du processus dans lequel il intervient.

*Exemple :* un web service qui renvoie les 10 derniers produits ajoutés au stock n’est intéressant que dans un cadre précis (par exemple pour afficher les nouveautés dans une page de la boutique), alors qu’un web service qui renvoie n’importe quel nombre de produits qu’on lui demande est potentiellement intéressant pour beaucoup d’autres web services. Par exemple l’ERP Comptabilité peut faire appel à ce web service pour récupérer l’ensemble des prix des produits dans le catalogue !

***Atomic*** :

Faites une seule chose et faites-la bien! Atomic veut dire indivisible et élémentaire. Évitez les fonctions dans vos web services qui par exemple passent les commandes, gèrent les retours, suppriment les produits et font le café. Évitez aussi une trop grande atomicité, on ne va pas créer un web service pour récupérer le nom d’un produit et l’autre sa description par exemple. Le tout est que celui-ci soit un bloc naturel et élémentaire.

*Exemple :* Un service qui a une fonction qui renvoie toutes les informations sur tous les grossistes et leurs commandes depuis la création de leurs comptes. Ce web service n'est clairement pas atomic : il sera composé d'une dizaines de composants qui iront chercher les infos des grossistes, les commandes, les prix, les infos sur les produits de chaque commandes, etc. Un web service atomic serait plutôt celui qui aura des fonctions séparées : une pour renvoyer la liste des grossistes, une autre la liste des commandes, etc. La logique étant que si nous voulons toutes ces informations, il suffit d’appeler chacune de ces fonctions et d'agréger les résultats.

#### **L'ESB pour centraliser les adaptateurs**

(Adaptateur (solution existante avant l’AOS) : utilisé lorsque l’application est composée de plusieurs sous-systèmes chacun utilisant son propre langage, l’adaptateur permet de faire la traduction et la communication entre les différents sous-systèmes)

Si vous voulez que votre **service 3** envoie à l'ERP comptabilité le résumé de chaque commande afin que les mouvements rentrent dans les comptes, il vous faudra communiquer avec lui via CSV/FTP. Or votre service communique uniquement en SOAP (via des fichiers XML). Donc la solution évidente est d'écrire du code qu'on appellera "adaptateur" qui générera un format CSV plutôt que XML. Puis vous l'enverrez via FTP à l'ERP en question.

ET HOP, voilà qui est résolu, d'autres questions ? :magicien:

Malheureusement ce n'est pas si facile que ça. Imaginez maintenant que vous êtes dans un SI plus grand et que vous devez communiquer avec une centaine de logiciels et services. Allez-vous écrire une centaines d'adaptateurs ? Les autres logiciels devront en faire autant ? Qu'en est-il des logiciels propriétaires qu'on ne peut pas modifier ? Vous allez certainement créer d'autres adaptateurs de votre côté pour comprendre leurS messageS, par exemple pour interpréter une requête via CSV reçue de l'ERP compta.

Tous cela commence à ressembler sérieusement à... un plat de spaghetti. ;)

#### L'ESB à la rescousse

Un Enterprise Service Bus (ESB) est un composant central qui se positionnera comme un interlocuteur unique pour tous les composants du SI. Ainsi, quand vous voudrez appeler un composant X, vous n'irez plus jamais lui parler directement. Vous ne saurez même pas quel protocole il accepte, ni son URL ! Il suffira d'envoyer une requête dans votre protocole préféré (dans notre cas SOAP) à l'ESB. Celui-ci s'occupera ensuite de faire le nécessaire pour transformer votre message et l'adapter avant de le transférer au composant demandé. Il fera ensuite la même chose pour la réponse avant de vous la servir sur un plateau d'argent, reste plus qu'à déguster !

Euh... l'ESB n'a pas de pouvoirs magiques, il faudra bien lui fournir des adaptateurs ?

Vous avez parfaitement raison, il faudra bien écrire des adaptateurs à fournir à l'ESB afin que celui-ci sache comment traduire les messages.

Sauf que l'énorme avantage ici, c'est qu'on n'aura plus une architecture en plat de spaghetti. Vos adaptateurs seront tous centralisés au même endroit. De plus vous n'avez plus besoin de dupliquer vos adaptateurs entre différents composants.

## Créez un web service Bottom-up

Vous revoilà pour développer votre premier web service.

Avant de commencer, il est impératif que vous ayez une bonne maitrise de WSDL, SOAP et leur fonctionnement. [**Je vous recommande de vous familiariser avec ces standards**](https://fr.wikipedia.org/wiki/Service_web). Je reviendrai ensuite sur ces concepts en les appliquant à notre web service.

### Les méthodes Top-down et Bottom-up

Il y a deux principales méthodes pour créer un web service.

1- **Top-down**

Il s'agit de la méthode la plus utilisée en entreprise. Cette méthode est aussi appelée Contract First. Il s'agit d'établir avant tout un contrat qui dit tout sur le fonctionnement du service avant de l'implémenter. Ce contrat n'est rien de plus qu'un fichier WSDL.

Vous établissez donc un fichier WSDL qui décrit votre service : les opérations possibles, les paramètres à passer et leurs types, etc.

Une fois ce contrat établi, vous créez le code nécessaire à son implémentation.

Rassurez-vous, vous n'aurez pas à écrire à la main une Class à partir d'un XML indigeste, il y a heureusement des moyens pour automatiser tout cela. En Java, la solution la plus populaire est JAX-WS. Nous y reviendrons en détails.

2- **Bottom-Up**

Il s'agit là de la manière la plus intuitive et facile pour créer un service. Vous créez votre code qui fait les opérations que vous voulez : dans notre cas une Class qui calcule le temps restant à vivre en fonction des paramètres passés. Puis vous générez automatiquement un WSDL à partir de cette Class ; là encore, JAX-WS le fera automatiquement.

Comme tout bon informaticien qui se respecte, nous allons commencer par le plus facile : Bottom-Up.

### Qu'est ce que JAX-WS ?

Java API for XML Web Services (JAX-WS) est une API de Java qui facilite la création de web services. Pour bien démystifier ce terme, sachez que JAX-WS est exactement la même chose que d'autres API que vous avez dû déjà utiliser comme java.lang (qui vous fournit justement String, Integer, etc.) , java.io (pour manipuler les fichiers, entre autres) ou encore java.sql.

JAX-WS va permettre de cacher entièrement la complexité des communications impliquant SOAP et WSDL.

* **Côté web service** : Quand un service externe appellera votre web service, il lui passera un simple fichier XML aux normes SOAP. Sans JAX-WS, vous devriez récupérer ce fichier et le parser pour essayer de comprendre quelles méthodes on cherche à invoquer et quels sont les paramètres passés. Je vous laisse deviner les centaines de lignes de code à écrire pour parser correctement un XML et faire appel aux bonnes méthodes dans les bonnes classes !

JAX-WS vous dispense de ce travail. Quand vous recevez un appel à votre web service, il va interpréter le message SOAP et appeler la bonne méthode dans votre code, exactement comme n'importe quelle méthode appellera une autre.

Mieux encore, quand vous retournerez une réponse, JAX-WS construira une réponse SOAP conforme et l'enverra au client.

De votre côté, vous n'y verrez que du feu. :p Vous vous contenterez de créer une Class comme celle qu'on a vu plus haut et y ajouter une annotation. JAX-WS fera le plus dur.

* **Côté Client** : Vous serez amenés à créer des clients pour appeler des web services dont vous ne connaissez rien. Vous n'aurez qu'un fichier WSDL complexe et long comme le bras pour comprendre comment les appeler, à quelle URL les trouver, quels paramètres passer et quel type de réponse recevoir. Vous imaginez bien qu'écrire du code pour parser un XML à chaque fois pour comprendre le fonctionnement d'un nouveau service est à la fois lent et fastidieux.

JAX-WS, là encore, vous dispense de tout ça. L'idée est simple : vous lui donnez l'emplacement du WSDL du nouveau service auquel vous voulez faire appel et il se charge de le parser et de **créer des classes Java** représentant le service en question. Vous n'avez plus qu'à appeler les classes créées comme vous appellerez n'importe quelle Class. Magique non ? :magicien:

Bien sûr, JAX-WS propose d'autres fonctionnalités. Mais ces deux-là sont les principales que nous allons aborder.

Passons à l'application !

**Annotations JAX-WS :**

* **@WebService** : Cette annotation dit simplement à JAX-WS que cette Class est à exposer en tant que web service. Elle peut prendre plusieurs paramètres, j'ai choisi de lui en passer un seul ici : serviceNname = "LifeLeft". Ce nom qu'on vient de donner à ce service servira ensuite pour accéder à notre service via son URL.
* **@WebMethod** : Cette annotation désigne cette méthode comme une méthode à exposer. Normalement si on ne précise pas cette annotation, toutes les méthodes de la Class seront exposées. Néanmoins, je vous recommande d'ajouter cette annotation à chaque méthode que vous voulez exposer, afin d'avoir la liberté d'ajouter d'autres méthodes supplémentaires pour votre usage interne, sans que celles-ci puissent être appelées par les services extérieurs.

#### **Modifier le pom.xml**

Dernière étape, nous allons ajouter 2 plugins à Maven. Voici le résultat :

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.lifeleft</groupId>

<artifactId>lifeleft</artifactId>

<packaging>war</packaging>

<version>1.0</version>

<name>lifeleft : combien vous reste-t-il à vivre?</name>

<build>

<finalName>lifeleft</finalName>

<plugins>

<plugin>

<artifactId>maven-war-plugin</artifactId>

<configuration>

<failOnMissingWebXml>false</failOnMissingWebXml>

</configuration>

</plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.7.0</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

</plugins>

</build>

</project>

* Vous précisez que le format que vous souhaitez est war grâce à war.
* Vous ajoutez le plugin maven-war-plugin qui permettra à Maven de déclencher les bonnes opérations qui aboutiront au packaging de votre service en .war.
* Vous ajoutez le plugin maven-compiler-plugin qui est simplement le compilateur Java de Maven.

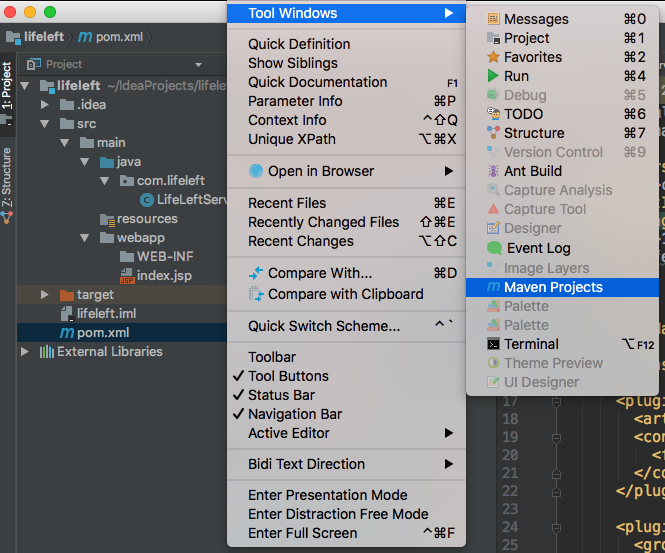
Vous avez maintenant tout ce qu'il faut pour packager votre application.

Vous pouvez bien sûr packager votre application via la ligne de commande suivante :

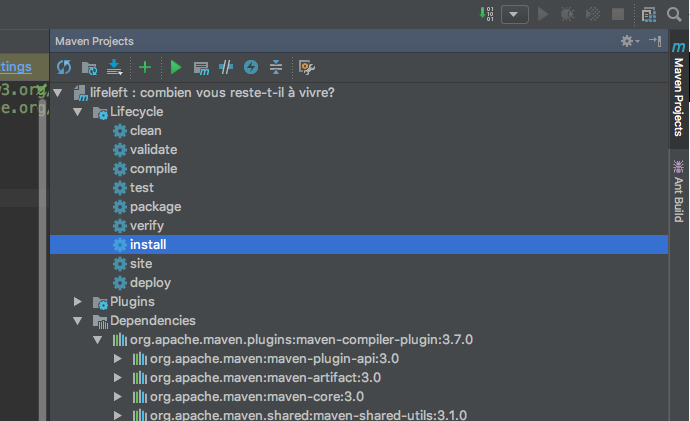
mvn install

Néanmoins, nous allons utiliser IntelliJ pour exécuter cette commande directement depuis l'interface graphique.

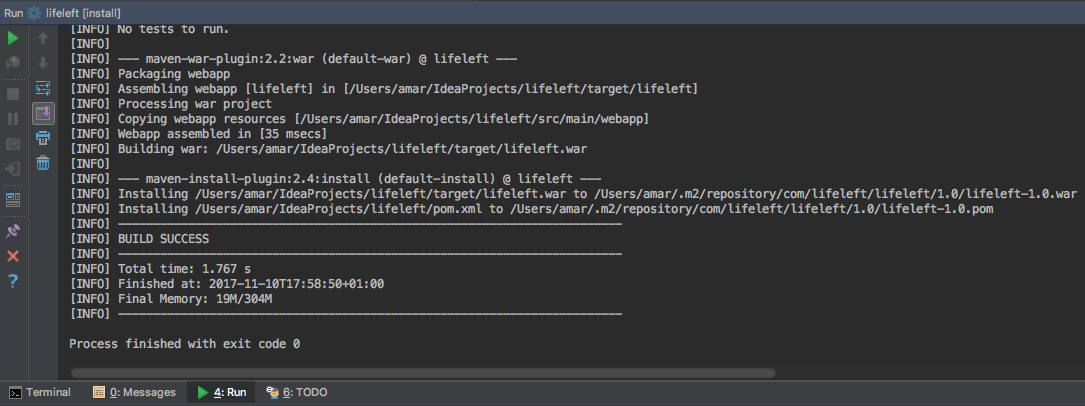
1. Sélectionnez *View > Tool Windows > Maven Projects*



2. Sélectionnez ensuite à gauche de la fenêtre Maven, puis déroulez *Lifecycle* et cliquez sur *Install*.



La commande est alors exécutée, et votre application est packagée ! Vous devriez avoir un message Maven indiquant le succès de la commande dans l'onglet *Run* en bas de la fenêtre.



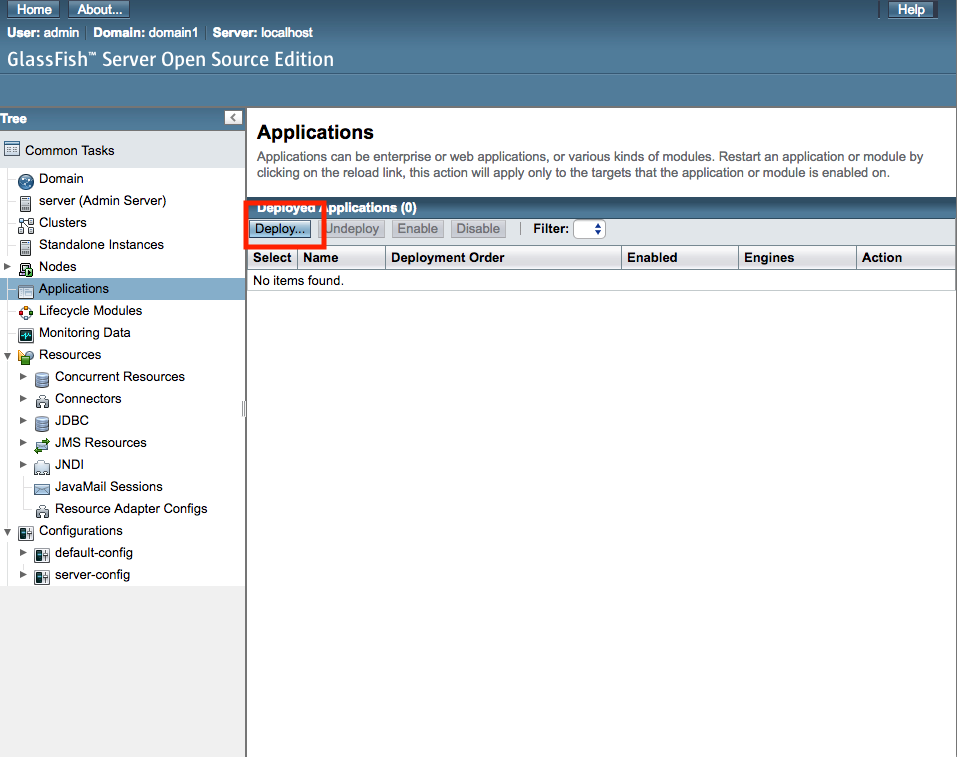
Vous pouvez vérifier la présence de votre service sous le nom *lifeleft.war* dans le nouveau dossier *Target* créé dans votre projet.

Très bien, vous avez maintenant tout ce qu'il vous faut pour lancer votre premier web service SOAP !

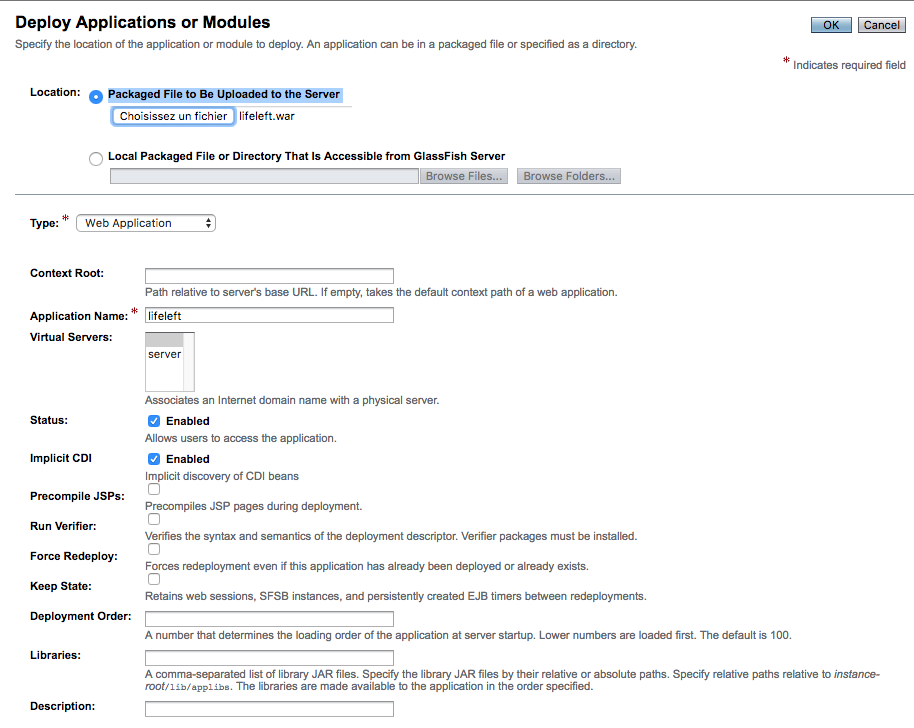
#### Déployer un service sur Glassfish

Normalement votre serveur Glassfish est démarré. Pour le vérifier, rendez-vous à http://localhost:4848. Si tel n'est pas le cas, je vous invite à le démarrer en suivant les étapes précédemment citées.

1- Cliquez sur Applications dans le panneau de gauche puis cliquez sur Deploy dans la nouvelle fenêtre.

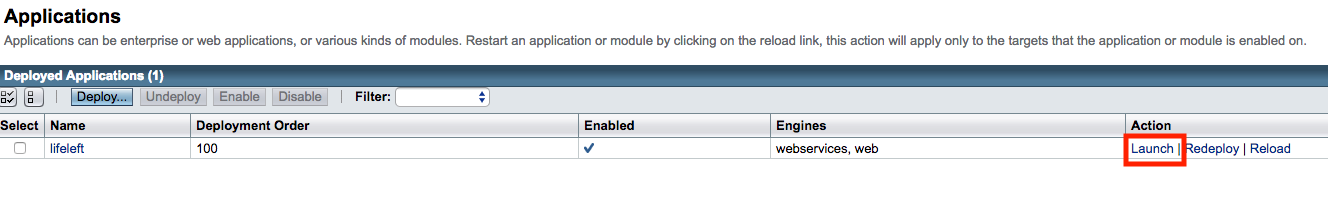


2- Sous Packaged File to Be Uploaded to the Server cliquez sur le bouton puis sélectionnez lifeleft.war sur votre ordinateur. Assurez-vous ensuite que Type: Web Application est choisi. Vous pouvez laisser tout le reste par défaut.



Cliquez sur OK.

3- Dans la nouvelle fenêtre, cliquez sur Launch en face de votre service.



4- Une page avec 2 liens s'ouvre : un lien http et l'autre en https. Sélectionnez le premier (http).

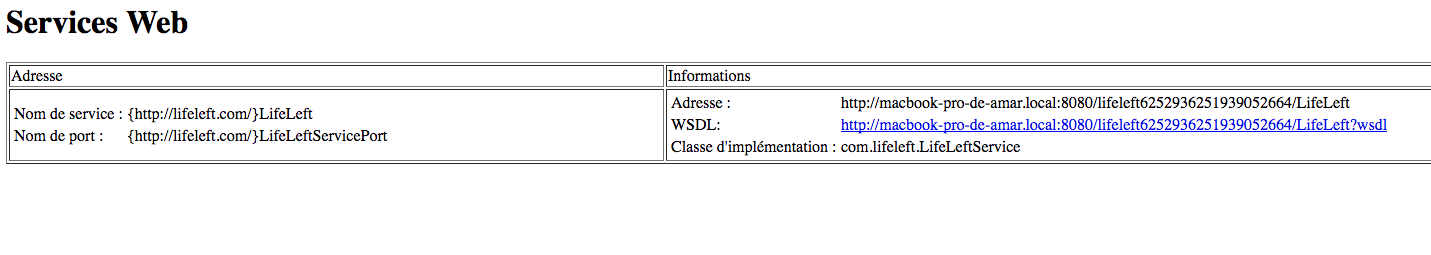
Vous devriez tomber sur une page avec le message "Hello World!". Ce message vient du fichier index.jsp généré par défaut avec votre projet Maven.

5- Pour accéder à votre service, ajoutez à votre URL LifeLeft. Celle-ci devrait alors ressembler à quelque-chose comme ça :

http://nomDeVotreOrdinateur.local:8080/lifeleft6252936251939052664/LifeLeft

Vous pouvez sans problème remplacer nomDeVotreOrdinateur.local par localhost.

Vous devriez tomber sur une page qui vous donne diverses informations sur votre service.



Glassfish génère automatiquement un WSDL qui décrit votre service à partir de votre code. Pour le consulter, cliquez sur le lien finissant par ?wsdl.

#### Testez votre service

Je vais survoler ici rapidement le test du service via les outils Glassfish, avant de revenir en détail sur un test complet d'un service via SoapUI.

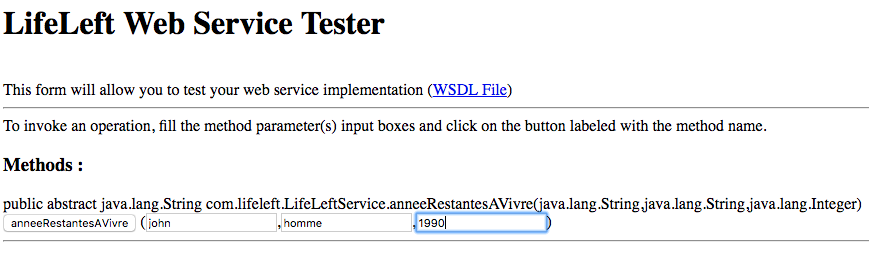
À présent que votre service tourne sur le serveur, vous allez pouvoir le tester très facilement, grâce encore une fois à Glassfish.

Pour ce faire, ajoutez ?Tester à votre URL. Elle devrait ressembler à ceci :

http://nomOrdinateur.local:8080/lifeleft6252936251939052664/LifeLeft?Tester

Vous retrouverez alors votre méthode précédemment créée avec 3 champs correspondant aux trois paramètres que vous avez défini.

Renseignez alors un prénom, un sexe et une date de naissance et validez.



Vous voilà avec une page présentant différentes informations que je vais commenter en plus de votre réponse. Notez que cette page est générée par Glassfish pour vous permettre de débugger et voir ce qui s'est passé lors de l'invocation de votre service. Pour vous faciliter la compréhension des messages échangés via SOAP, [je vous rappelle le début de ce cours, à consulter ici.](https://openclassrooms.com/courses/les-services-web#/id/r-218997) .

1. ***Method parameter(s)*** ici on reprend simplement la liste des paramètres passés.
2. ***Method returned*** ici vous retrouvez le fruit de votre dur labeur! La réponse retournée par votre service. Assurez-vous que le résultat est correct.
3. ***SOAP Request*** il s'agit de la requête SOAP que Glassfish a généré en se basant sur le WSDL et les paramètres que vous avez renseigné.
4. <?xml version="1.0" encoding="UTF-8"?>
5. <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/" xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
6. <SOAP-ENV:Header/>
7. <S:Body xmlns:ns2="http://lifeleft.com/">
8. <ns2:anneeRestantesAVivre>
9. <arg0>john</arg0>
10. <arg1>homme</arg1>
11. <arg2>1990</arg2>
12. </ns2:anneeRestantesAVivre>
13. </S:Body>
14. </S:Envelope>
    * Envelope : l'enveloppe du message SOAP qui délimite le début et la fin de celui-ci. Elle définit les namespaces utilisés : S et SOAP-ENV.
    * Header ici le header optionnel est vide.
    * Body : qui définit le corps du message, déclare un namespace ns2 pour identifier votre méthode anneeRestantesAVivre et les 3 paramètres qu'on lui passe.
15. ***SOAP Response***
16. <?xml version="1.0" encoding="UTF-8"?><S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/" xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
17. <SOAP-ENV:Header/>
18. <S:Body xmlns:ns2="http://lifeleft.com/">
19. <ns2:anneeRestantesAVivreResponse>
20. <return>Bonjour john, il vous reste 52 ans à vivre, profitez-en au maximum !</return>
21. </ns2:anneeRestantesAVivreResponse>
22. </S:Body>
23. </S:Envelope>

Ceci est la fameuse réponse générée par JAX-WS. Elle a la même structure que la requête, à la différence qu'ici dans le Body nous avons la réponse obtenue. Vous remarquerez que l'on a suffixé le nom de votre méthode par Response.

## Créez un client et un web service Top-down

Nous allons nous attaquer ici à la méthode la plus utilisée pour la création de web services en entreprise. Cette méthode consiste à créer un contrat avant d'écrire le code. Le contrat est bien sûr un fichier WSDL qui décrit en détail votre service.

Cette approche a l'avantage de vous permettre d'exposer à tous le monde (et donc aux futurs clients) les caractéristiques de votre service pour validation. Vous pourrez alors répondre à des questions comme :

* Est-ce que mon service correspond à ce qui a été demandé ?
* Est-ce que les réponses que mon service renvoie sont satisfaisantes ?
* Manque-t-il des opérations ?
* Certaines opérations sont-elles déjà proposées par d'autres services ?
* etc.

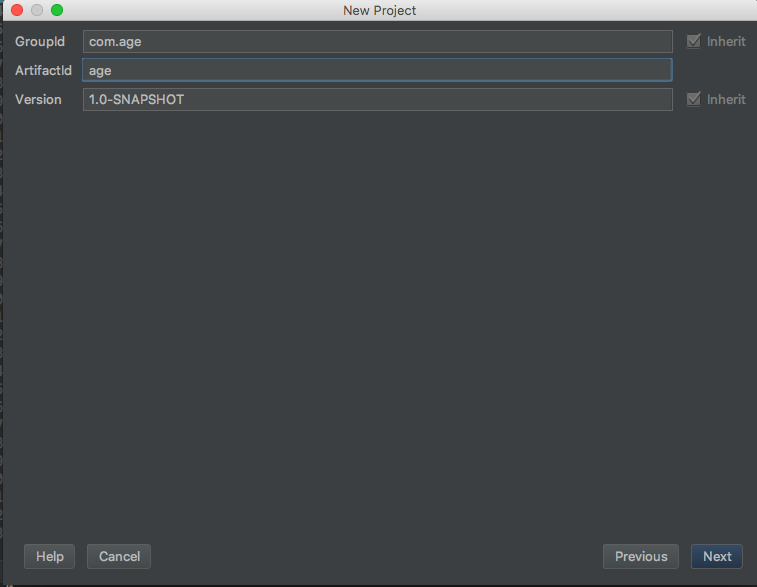
Car il faut savoir qu'un web service est en général plus complexe que celui que vous avez créé précédemment, il est donc préférable de valider d'abord, plutôt que de revenir sur un code complexe pour le remanier entièrement.

Cette méthode est aussi un passage obligé quand vous voudrez créer un client pour votre service. Vous n'aurez à ce moment-là rien d'autre qu'un grand WSDL indigeste. Hors de question de passer des heures à le décortiquer pour écrire le bon code qui enverra les bonne requêtes au bon format aux bonnes adresses. Nous allons voir comment générer automatiquement un client grâce à wsimport à partir d'un WSDL.

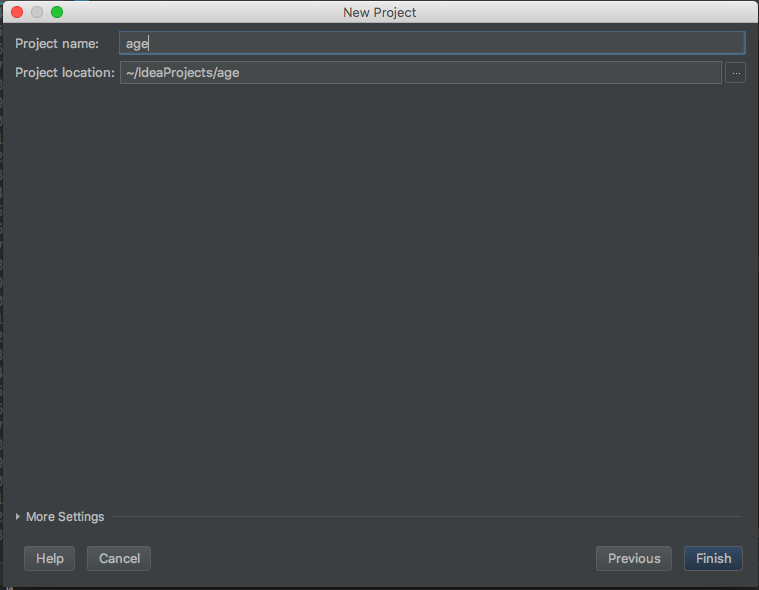
### Mise en place du web service

Les étapes ici sont les mêmes que lorsque vous avez créé le premier web service avec la méthode Bottom-Up dans IntelliJ IDEA.

Vous allez simplement remplacer le GroupeID et l'artifactId par com.age et age.

Figure 29

Vous allez aussi nommer le projet age à la dernière étape.

Figure 30

Une fois votre projet créé, modifiez le pom.xml afin qu'il soit le même que pour le projet précédent.

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.age</groupId>

<artifactId>age</artifactId>

<packaging>war</packaging>

<version>1.0</version>

<name>Pour connaître ton âge plus besoin d'envoyer "âge" au 8 12 12 ! </name>

<build>

<finalName>age</finalName>

<plugins>

<plugin>

<artifactId>maven-war-plugin</artifactId>

<configuration>

<failOnMissingWebXml>false</failOnMissingWebXml>

</configuration>

</plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.7.0</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

<plugin>

<groupId>org.codehaus.mojo</groupId>

<artifactId>jaxws-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

Pensez à supprimer le fichier **web.xml** sous WEB-INF.

**N'oubliez pas sur IntelliJ de marquer le dossier java comme source et de créer la Class vide AgeService sous le dossier java.**

### Créer un fichier WSDL

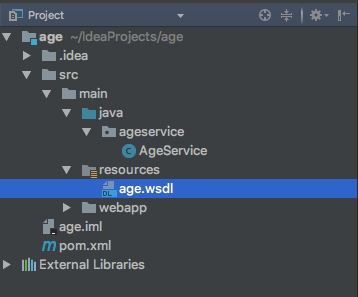
Si ce n'est pas déjà fait, n'oubliez pas de consulter [**ce chapitre**](http://www.pfl-cepia.inra.fr/uploads/images/GestionDonneesImages/unites/GdpDoc/OpenClassrooms-servicesWeb.pdf).

La première chose est donc la création d'un fichier WSDL qui décrit notre service.

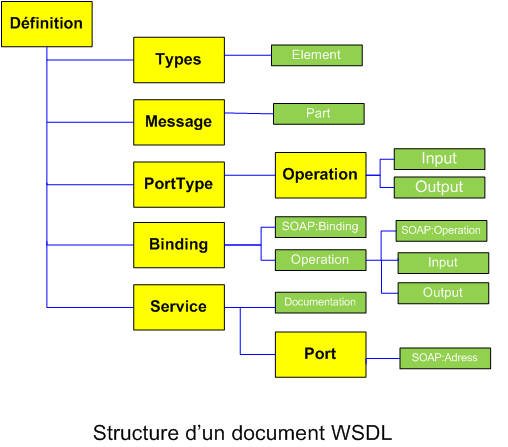
Dans la vie réelle, vous n'aurez pratiquement jamais à en créer à la main, des outils graphiques disponibles sur le marché ou dans les IDE permettent d'en créer, par exemple : [WSDL Editor](https://wiki.eclipse.org/Introduction_to_the_WSDL_Editor) d'Eclipse ou encore [Liquid](https://www.liquid-technologies.com/xml-schema-editor" \t "_blank).

Néanmoins, nous allons ici en créer un à la main afin que vous vous familiarisiez avec ce format. Sachez aussi que vous aurez, tôt ou tard, affaire à ce fichier car il est nécessaire de le comprendre pour débugger dans certains cas.

* Dans le dossier resources, créez un fichier et nommez-le age.wsdl.
* Dans le dossier java créez un Package et nommez le ageservice.
* Créez ensuite dans le Package ageservice une Class java nommée AgeService.

Figure 31

Pour rappel, voici la [structure d'un document WSDL](https://openclassrooms.com/courses/les-services-web#/id/r-218997) :

Figure 32

Dans le fichier age.wsdl :

* **Créez la balise < definitions > comme suit :**

<?xml version="1.0" encoding="UTF-8"?>

<definitions

xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"

targetNamespace="http://ageservice/"

xmlns:tns="http://ageservice/"

xmlns="http://schemas.xmlsoap.org/wsdl/"

name="AgeService">

La balise definitions est la racine de tout WSDL, c'est-à-dire la balise qui va englober toutes les autres, elle définit les espaces de noms (namespaces) utilisés dans le document.

Ici, nous avons déclaré :

* soap : définit des conventions dans le nommage et la structure de certains éléments comme body ou operation que nous verrons plus tard. Si vous voulez voir les éléments concernés, il suffit de vous rendre à l'URL indiquée.
* targetNamespace : ceci est votre espace de nom qui fait référence à ce document même. Il est à noter que dans ce cas, on peut mettre n'importe quelle valeur, vous n'êtes pas obligé de mettre une vraie URL. Cela peut tout à fait être remplacé par un GUID ([générateur GUID](https://www.jetbrains.com/idea/download/)) par exemple xmlns:tns="cdf382d1-904d-4f7b-ab43-144623206c3a". D'ailleurs vous remarquerez que j'ai choisi un namescpace fictif http://ageservice/ qui n'est pas forcement une URL valide.
* tns : c'est la même chose que targetNamespace, la différence c'est qu'on attribue un préfixe tns à ce namespace qui nous permettra plus tard d'y faire référence pour indiquer que tel tag appartient à notre document. Avec targetNamespace ceci n'est pas possible en absence justement de préfixe. Il est recommandé que targetNamespace et tns soient les mêmes.
* xmlns : ceci est le namespace par défaut si jamais il y a un élément qui n'appartient à aucun.
* name : un nom que vous donnez à votre service.
  + **Créez la balise < types >**

<types>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" targetNamespace="http://ageservice/">

<xs:element name="getAge" type="tns:getAge"/>

<xs:element name="getAgeResponse" type="tns:getAgeResponse"/>

<xs:complexType name="getAge">

<xs:sequence>

<xs:element name="arg0" type="xs:int" minOccurs="0"/>

</xs:sequence>

</xs:complexType>

<xs:complexType name="getAgeResponse">

<xs:sequence>

<xs:element name="return" type="xs:string" minOccurs="0"/>

</xs:sequence>

</xs:complexType>

</xs:schema>

</types>

Ici vous allez définir les types de données qui vont transiter entre le client et votre service. WSDL ne définissant pas un typage limité, libre à vous de créer vos propres types grâce à la balise complexType.

* Nous définissons un namespace xmlns:xs pour isoler dans celui-ci les balises que nous allons créer. Le but là encore est qu'il n'y ait pas de confusion dans les noms avec d'autres balises qu'on pourrait créer plus tard. Il sert aussi à montrer que notre structure doit correspondre à celle définie dans http://www.w3.org/2001/XMLSchem, ce qui permet plus tard la validation de éléments.
* xs:element : nous définissons un élément getAge et un autre getAgeResponse.
* xs:sequence : cette balise indique que les éléments qui composent ce complexType doivent apparaitre dans l'ordre. C'est la balise la plus utilisée car souvent quand on souhaite recevoir des paramètres à passer à une fonction, celles-ci doivent être dans l'ordre. Pour rappel les deux autres possibilités sont :
  + xs:all : l'ordre n'est pas important
  + xs:choice : un seul élément parmi une liste proposée est accepté.
* xs:complexType nous revenons ici sur chaque élément pour détailler sa structure et son type de contenu. Ainsi pour getAge il est composé d'un int qu'on appelle arg0. minOccurs="0" (minimum occurrence) indique que cet élément est facultatif.

Ici nous n'avons aucun vrai type complexe pour justifier leur création, nous n'avons besoin que d'un int en paramètre d'entrée. Néanmoins j'ai enveloppé ça dans un type complexe car c'est sous ce format que vous trouverez les types dans les WSDL la plupart du temps, notamment parce qu'ils sont souvent générés automatiquement.

Au final ici nous avons un peu réinventé la roue on définissant un élément int qui sera plus tard utilisé en entrée de notre méthode qui calculera l'âge et un élément string qui sera la phrase retournée par celle-ci.

* **Créez la balise < message >**

<message name="getAge">

<part name="parameters" element="tns:getAge"/>

</message>

<message name="getAgeResponse">

<part name="parameters" element="tns:getAgeResponse"/>

</message>

Les messages indiquent les paramètres qui seront passés plus tard aux opérations (les opérations étant l'équivalent des méthodes dans notre Class). Ne pas confondre donc message et type : type définit la nature d'un élément qu'on pourra utiliser comme on veut, alors que message fixe réellement lequel de ces éléments déclarés sera passé aux opérations.

* **Créez la balise < portType >**

<portType name="AgeService">

<operation name="getAge">

<input message="tns:getAge"/>

<output message="tns:getAgeResponse"/>

</operation>

</portType>

Ici nous indiquons enfin nos opérations possibles. Cela revient à indiquer quelles méthodes seront disponibles dans notre service et accessibles de l'extérieur. C'est l'équivalent de @WebMethod qu'on a vu plus tôt.

* operation : nom qu'on souhaite donner à la méthode qui sera générée.
* input : paramètre d'entrée de notre méthode, ici nous avons utilisé getAge qu'on a défini plus tôt.
* Vous pouvez imaginer le potentiel des complexType, car imaginez que vous ayez besoin d'un type **Client** qui contient toutes les informations d'un client (nom, adresse, liste d'achats) pour pouvoir l'ajouter à une base de données. Dans un programme Java classique, vous créerez une Class Client avec tout ce qu'il faut. Dans notre cas, c'est un service extérieur qui fait appel à vous et il est peut être écrit en C++, Python ou autre. Grâce au complexType il va pouvoir vous communiquer le type Client que vous avez défini, un peu comme s'il remplissait un formulaire, que vous récupérerez ensuite, sous forme d'une instance d'une Class client grâce à JAX-WS.
  + output : paramètre de sortie.
  + **Créez la balise < binding >**

<binding name="AgeServicePortBinding" type="tns:AgeService">

<soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>

<operation name="getAge">

<input>

<soap:body use="literal"/>

</input>

<output>

<soap:body use="literal"/>

</output>

</operation>

</binding>

* binding décrit concrètement comment les opérations définies dans portType seront invoquées sur le réseau.
* soap:binding est utilisée pour indiquer que la communication se fera via le protocole SOAP.
* transport="http://schemas.xmlsoap.org/soap/http" indique que les messages SOAP seront transmis via HTTP.
* On indique enfin que pour l'opération getAge qu'on a définie dans portType, les paramètres en input et output seront encodés en literal, c'est-à-dire qu'ils seront mis en forme selon les définitions que nous avons spécifiées pour structurer les messages SOAP qu'on va renvoyer.
  + **Créez la balise < service >**

<service name="getAge">

<port name="AgeServicePort" binding="tns:AgeServicePortBinding">

<soap:address location="http://localhost:8080/Age/getAge"/>

</port>

</service>

Cette dernière balise indique littéralement l'URL pour appeler les opérations que nous avons définies. Dans ce cas, nous n'avons qu'une opération qui se trouve à http://localhost:8080/Age/getAge.

* age.wsdl final

<?xml version="1.0" encoding="UTF-8"?>

<definitions

xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"

targetNamespace="http://ageservice/"

xmlns:tns="http://ageservice/"

xmlns="http://schemas.xmlsoap.org/wsdl/"

name="AgeService">

<types>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" targetNamespace="http://ageservice/">

<xs:element name="getAge" type="tns:getAge"/>

<xs:element name="getAgeResponse" type="tns:getAgeResponse"/>

<xs:complexType name="getAge">

<xs:sequence>

<xs:element name="arg0" type="xs:int" minOccurs="0"/>

</xs:sequence>

</xs:complexType>

<xs:complexType name="getAgeResponse">

<xs:sequence>

<xs:element name="return" type="xs:string" minOccurs="0"/>

</xs:sequence>

</xs:complexType>

</xs:schema>

</types>

<message name="getAge">

<part name="parameters" element="tns:getAge"/>

</message>

<message name="getAgeResponse">

<part name="parameters" element="tns:getAgeResponse"/>

</message>

<portType name="AgeService">

<operation name="getAge">

<input message="tns:getAge"/>

<output message="tns:getAgeResponse"/>

</operation>

</portType>

<binding name="AgeServicePortBinding" type="tns:AgeService">

<soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>

<operation name="getAge">

<input>

<soap:body use="literal"/>

</input>

<output>

<soap:body use="literal"/>

</output>

</operation>

</binding>

<service name="getAge">

<port name="AgeServicePort" binding="tns:AgeServicePortBinding">

<soap:address location="http://localhost:8080/Age/getAge"/>

</port>

</service>

</definitions>

Maintenant que nous avons notre contrat, nous allons pouvoir utiliser l'outil wsimport pour générer toutes les classes nécessaires à l'implémentation de notre service.

### Générer des artefacts AJAX-WS grâce à wsimport

wsimport est un outil inclus dans le JDK qui permet de générer les artefacts JAX-WS nécessaires pour créer un client ou un service à partir d'un WSDL. Ces artefacts sont tout simplement des classes et des interfaces qui font le boulot que nous avons fait plus tôt dans la démarche Bottom-up.

#### Utiliser wsimport

Si Java est installé, alors wsimport est aussi disponible ! Pour le vérifier, il suffit d'entrer la commande suivante :

wsimport -version

Résultat

wsimport version "2.2.9"

* Les documents WSDL sont censés être dans un UDDI (vous vous souvenez ? :diable: ). Afin de s'affranchir des problèmes des chemins vers le WSDL, nous allons le placer dans un répertoire Github dont nous allons récupérer l'URL directe.
* Commencez par vous placer dans le dossier age du projet.

cd /Chemin/Vers/Dossier/IdeaProjects/age

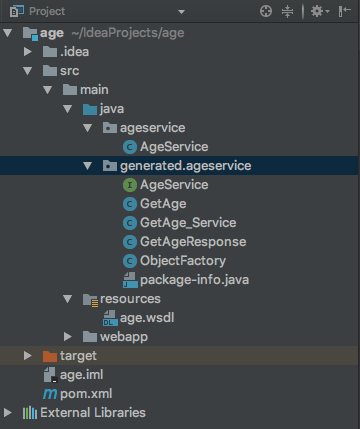
* Générez les artefacts comme suit :

wsimport -Xnocompile -d ./src/main/java -p generated.ageservice https://raw.githubusercontent.com/AdamSoufiane/age/master/age.wsdl

* -Xnocompile : indique que vous ne souhaitez pas que les Class générées soient compilées. En effet, nous les compilerons nous-mêmes grâce à Maven.
* -d : indique le dossier dans le projet dans lequel vous souhaitez mettre les Class qui seront générées.
* -p : indique à quel packet ces Class appartiendront. Dans ce cas, vous créez un packet spécialement pour celles-ci, appelé generated.ageservice.

Pour connaître les autres paramètres, [rendez vous dans la documentation](https://docs.oracle.com/javase/6/docs/technotes/tools/share/wsimport.html" \l "Example" \t "_blank).

Vous devriez obtenir cette arborescence :

Figure 33

#### Les fichiers générés

* AgeService : il s'agit ici d'une interface qui représente ce que nous devrions implémenter. En d'autres termes, si vous ouvrez le fichier, vous verrez que cette interface définit une méthode getAge qui accepte comme argument arg0 que nous avons défini et retourne un String. Cette interface va nous permettre de créer l'implémentation dans laquelle nous allons définir la logique de notre service (dans ce cas le calcul de l'âge à partir d'une date de naissance).
* GetAge et GetAgeResponse : ces Class représentent les complexType que nous avons créés précédemment dans la balise < types >. Ici elles n'ont pas une grande utilité, mais rappelez-vous de l'exemple que je vous ai donné avec la Class Client et nombreux attributs. Si nous avions déclaré un complexType qui définit un client avec tous ses attributs, nous aurions eu alors une belle Class Client.java avec toutes les méthodes et attributs prêts à l'emploi.
* GetAge\_Service : je vous parlerai de cette Class quand nous construirons le Client.
* ObjectFactory : une Factory pour créer des instances des différentes Class générées à partir des complexType. Dans notre cas il s'agit des Class getAge et GetAgeResponse.
* package-info.java : ce fichier permet en général de générer de la documentation pour les Packages en ajoutant des commentaires au-dessus des noms des Packages, mais dans notre cas sa principale fonction est de créer une association entre le Namespace http://ageservice/ que nous avions défini dans le WDSL et le package auquel il est associé dans les Class générées, à savoir generated.ageservice.

**Créer l'implémentation**

L'implémentation est très simple, il nous suffira d'implémenter l'Interface AgeService

package ageservice;

import javax.jws.WebMethod;

import javax.jws.WebService;

import java.math.BigInteger;

@WebService(serviceName = "getAge")

public class AgeService implements generated.ageservice.AgeService {

@WebMethod

public String getAge(Integer naissance) {

return "Votre âge est de " + (2017-naissance) + "ans";

}

}

C'est tout ! Vous venez de créer un web service en partant d'un WSDL. Évidemment, ici vous ne voyez pas forcement tous les avantages, car le code est court.

Mais imaginez que votre service gère des commandes et propose une dizaine de méthodes exposées et une dizaine de Class de type : Client, Commande, Paiement, chacun avec des attributs et méthodes. Dans ce cas-là, la puissance de wsimport apparaît clairement. En partant d'un WSDL, vous aurez toutes ces Class prêtes à l'emploi ainsi qu'un SEI formaté et annoté.

**Packagez et changez l'URL de votre web service**

Pour packager votre web service, il vous suffit de suivre les mêmes étapes que dans le service LifeLeft créé en Bottom-up. Pour rappel sur IntelliJ, il vous suffit de cliquer sur Install sous Lifecycle du panneau Maven.

Uploadez ensuite votre war dans Glassfish.

Vous avez dû remarquer que nous avons fixé l'URL sous la balise < service > dans notre WSDL à http://localhost:8080/Age/getAge. Or si vous vous rendez dans Applications sur votre serveur puis Launch vous remarquerez que votre service a une URL de ce type :

http://localhost:8080/age146146464161484/

Les chiffres après age sont un identifiant unique généré par Glassfish afin d'éviter que vous ayez deux services avec le même nom.

Comme nous ne souhaitons pas de chiffres dans notre URL afin qu'elle soit conforme à notre WSDL, nous allons les supprimer.

Rendez-vous dans Applications puis cliquez sur le nom de votre service age dans la colonne Name.

Sous Context Root: remplacez le tout par **/Age** puis validez.

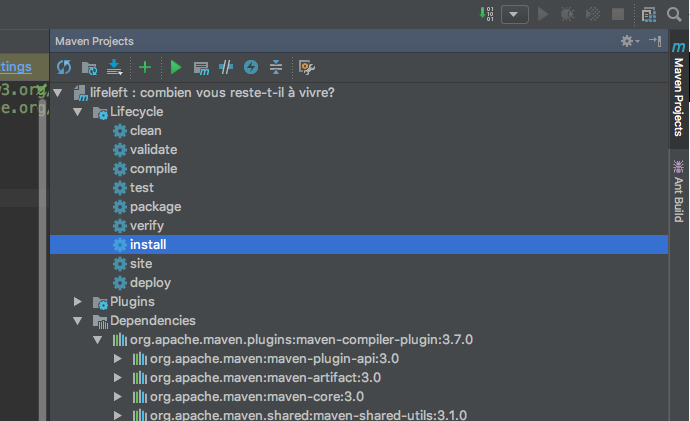
**Testez votre web service**

Pour tester votre web service, vous pouvez procéder de la même manière que pour LeftLife.

Si vous avez suivi les mêmes noms que dans les exemples que je vous ai donnés, vous devriez avoir une URL de test comme celle-ci :

http://localhost:8080/Age/getAge?Tester

Entrez ensuite une date de naissance, vous devriez alors avoir une réponse comme celle-ci

Figure 34

### Créer un client à partir d'un WSDL

Vous allez à présent créer un client pour votre web service LifeLeft.

Commencez par créer un autre projet vide exactement comme vous aviez fait avec LifeLeft.

GroupeId : com.leftlifeclient ArtifactId : leftlifeclient Project name : leftlifeclient

**Ne créez par contre aucune Class.**

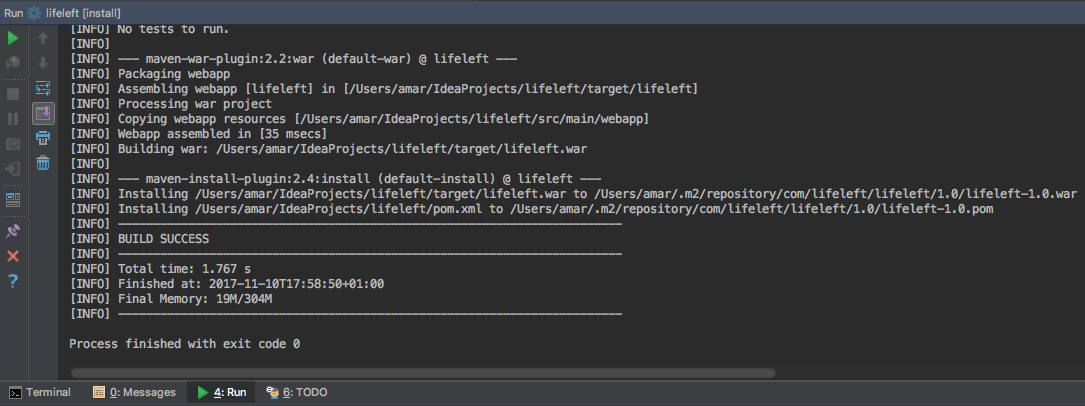
* Récupérez le WSDL de LifeLeft depuis Glassfish. Pour ce faire, allez dans Applications puis cliquez sur Launch en face de LifeLeft et cliquez sur le premier lien avec le port 8080.

Ajoutez le nom de service que vous aviez défini dans serviceName de @WebService. Cela devrait vous donner une URL comme celle-ci :

http://nomDeVotreOrdinateur.local:8080/lifeleft/LifeLeft

**Remarquez** que j'ai changé l'URL de ce service également pour ne garder que lifeleft au lieu de lifeleft45656426146. Je vous invite à faire de même.

Dans la page qui s'ouvre, vous avez l'adresse de votre WSDL à droite.

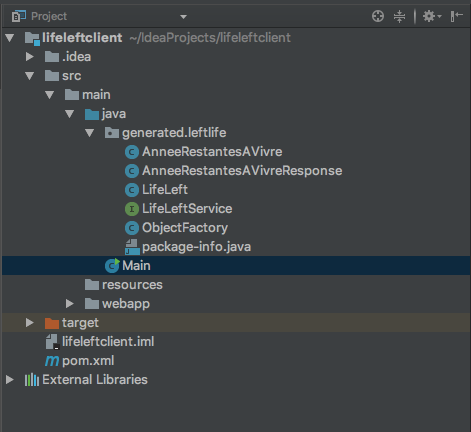
Figure 35

C'est cette adresse qui va nous permettre de générer notre client à partir du WSDL.

* Comme nous l'avons fait plus tôt, placez-vous dans le dossier lifeleftclient depuis la ligne de commande.
* Générez le client :
* wsimport -Xnocompile -d ./src/main/java -p generated.leftlife http://localhost:8080/lifeleft/LifeLeft?wsdl

On utilise la même commande que précédemment, mais cette fois nous allons nous intéresser à l'implémentation d'un client à partir des Class générées.

Vous obtenez alors ceci :

Figure 36



Regardez la Class LifeLeft. Cette classe est appelée une Class proxy. Elle va vous permettre d'appeler le service distant LifeLeft comme s'il était local. JAX-WS vous cache tout ce qui concerne la formulation des requêtes SOAP, leur envoi via le protocole choisi (ici HTTP) et l'interprétation des réponses.

L'annotation @WebServiceClient est une annotation utilisée par les générateurs comme wsimport pour localiser < service > dans le WSDL. Vous n'aurez pratiquement jamais à utiliser cette annotation.

Cette Class hérite de la Class Service qui est une Class capable de fournir une représentation en Java d'un service à partir de son WSDL.

S'ensuivent plusieurs définitions de constantes déclarant les URL et Namespaces qu'on a déclaré dans le WSDL.

Ce qui nous intéresse ici est principalement la méthode getLifeLeftServicePort qui va nous donner accès à toutes les opérations déclarées dans le WSDL. Dans notre cas il y a une seule opération : anneesRestantesAVivre.

* Utiliser le client

Nous allons maintenant invoquer le code généré pour appeler notre service distant LifeLeft comme s'il était un simple package dans notre code.

Créez une Class Main sous le dossier java.

Invoquez le service :

import generated.leftlife.LifeLeft;

import generated.leftlife.LifeLeftService;

public class Main {

public static void main(String[] args) {

LifeLeft lifeleft = new LifeLeft();

LifeLeftService lifeleftsrv = lifeleft.getLifeLeftServicePort();

String resultat = lifeleftsrv.anneeRestantesAVivre("john", "homme", 1980);

System.out.println(resultat);

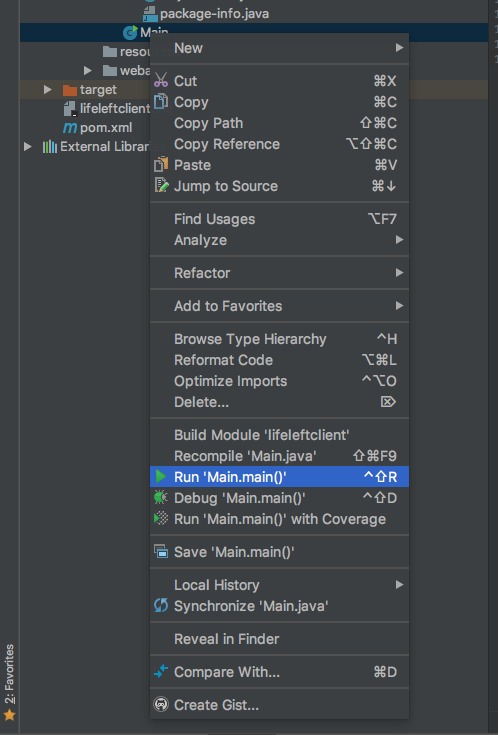
} }

* + **Ligne 8** : On crée une instance de la Class LifeLeft générée.
  + **Ligne 10** : On demande une instance LifeLeftService qui contient l'opération anneesRestantesAVivre.
  + **Ligne 11** : Nous appelons la méthode anneesRestantesAVivre en lui passant les paramètres nécessaires. C'est à ce moment-là que JAX-WS fera le nécessaire pour formuler une requête SOAP et l'envoyer à votre web service LifeLeft puis recevoir la réponse SOAP également et en extraire le résultat sous forme de String.
  + **Ligne 13** : Écrire le résultat dans la sortie standard.
* Tester le client

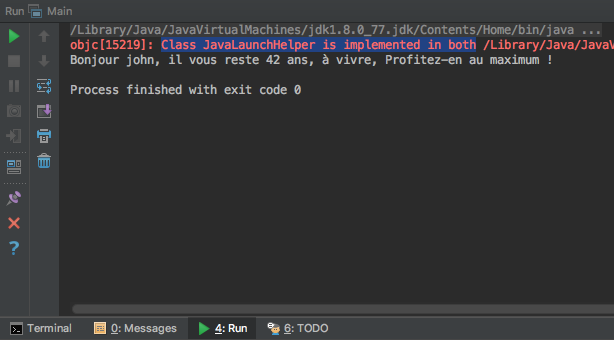
Comme notre lifeleftcient n'est pas forcément un web service, vu qu'il ne propose aucune interaction avec le monde extérieur, nous n'avons pas besoin de le déployer dans un serveur comme Glassfish.

Nous allons tout simplement le compiler et l'exécuter en local.

Pour ce faire, faites un clic droit sur le main, puis cliquez sur Run 'Main.main()'.

Figure 37

* Vous devriez avoir alors une réponse de ce type :

Figure 38

Si jamais vous avez dans la sortie sur IntelliJ un message d'erreur comme "objc[15219]: Class JavaLaunchHelper is implemented in both ... ", sachez que ce n'est qu'un ancien bug spécifique à IntelliJ, il n'a aucun impact sur votre code.

Voilà ! Vous venez de recevoir une réponse de votre web service. Vous n'avez eu à écrire aucune ligne de XML ni lire aucun WSDL.

Tout ça est très bien, mais vous ne trouvez pas que c'est un peu opaque ? On aimerait quand même voir ce qui se passe, quel message est envoyé et quel message est reçu, surtout en cas de bug !

Je vais donc vous apprendre dans la partie suivante à utiliser un out