

VINCI THERMOGREEN PROGRAMMER HANDBOOK

Page de service

Référence : N/A

Plan de classement : stadium-technic-analyse-conception-thermo-green

Niveau de confidentialité : Corporate

Mises à jour :

VERSION	DATE	AUTEUR	DESCRIPTION
1.0	21/01/2023	Thomas PRADEAU	Création du document
1.1	01/11/2023	Thomas PRADEAU	Mise à jour de l'analyse conception
1.2	02/11/2023	Thomas PRADEAU	Ajouts authentification et gestion des erreurs.
1.3	20/11/2023	Thomas PRADEAU	Gestion des alertes SMS.
1.4	30/12/2023	Thomas PRADEAU	Ajout signature et certification archive JAR.

Validations :

VERSION	DATE	NOM	RÔLE
1.3	20/11/2023	Jérôme VALENTI	Enseignant

Diffusions :

VERSION	DATE	NOM	CADRE DE LA DIFFUSION
---------	------	-----	-----------------------

Sommaire

Table des matières

Page de service	1
Sommaire	2
Analyse conception	4
Définition des besoins	4
Modélisation des classes métier	5
Diagramme d'expérience utilisateur	5
Diagramme de séquence « boîte noire »	6
Diagramme de séquence	7
Conception	8
Architecture	8
Implémentation des classes métier	9
Classe modèle mesure	10
Classe modèle stade	11
Classe modèle utilisateur	12
Implémentation des vues	13
Vue console	14
Vue de connexion	15
Vue de changement de mot de passe	16
WindowBuilder	17
Implémentation du contrôleur	18
Le contrôleur	18
Fonctionnalités du contrôleur	19
Persistance des données	22
MySQL	22
Connexion avec la base de données	24
La classe DatabaseHelper	25
Données de connexion	26
Authentification	28
Hash du mot de passe	29
Implémentation	29
Force et prérequis d'un mot de passe	30
Validation du mot de passe	31
Internationalisation	32
Implémentation	32
Gestion des erreurs	34
Implémentation	34

Exemples de cas d'erreur	35
Journalisation.....	35
Gestion des messages d'alerte	36
API Vonage	36
Client Java Vonage	37
Implémentation.....	37
Documentation et déploiement	39
Documentation.....	39
Déploiement.....	41
Bibliographie.....	49
Table des illustrations	50

Analyse conception

Définition des besoins

L'entreprise Vinci travaille dans la construction et la gestion de stades. Parmi les besoins associés, on y trouve la gestion de la pelouse présente dans les différents stades.

Les stades sont équipés de détecteurs de température ainsi que d'arroseurs automatiques. L'objectif serait alors de centraliser les informations rapportées des capteurs et de faciliter la gestion de plusieurs stades au travers d'une solution applicative permettant aux employés habilités de travailler facilement sur les stades et d'intervenir en cas de besoins.

Les capteurs présents dans le stade sont distingués parmi différentes zones du stade, la solution applicative devra donc distinguer les mesures de température selon le stade, l'heure et la zone de celui-ci, sachant que chaque zone est relative au stade correspondant.

Les informations seront stockées dans une base de données, accessible par la solution applicative, de manière à centraliser les informations. On trouvera par ailleurs les agents de gestions de Vinci, ainsi que les stades enregistrés et gérés par la solution.

Une solution d'authentification et de chiffrement sera mise en place afin de garantir l'intégrité et la confidentialité des données gérées et exploitées par Vinci en rapport avec la gestion des stades.

Dans le cas où des températures relevée seraient en dehors des limites posées, le personnel en charge du stade doit être en mesure d'envoyer une notification par SMS au numéro de téléphone du personnel de maintenance.



Modélisation des classes métier

Le diagramme suivant représente la modélisation des différentes classes métier :

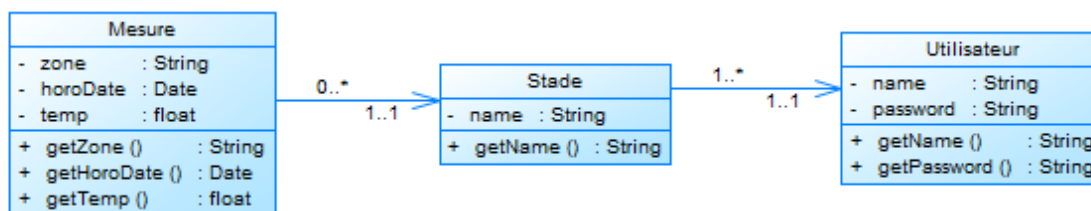


Figure 1 : Modélisation des classes métier

Dans ce diagramme de classes, on retrouve la classe utilisateur possédant un nom et un mot de passe, la classe stade identifiée par un nom, ainsi que la classe mesure correspondant aux mesures effectuées dans les stades comportant la zone du stade en question, la date et l'heure ainsi que la valeur enregistrée.

On considère un utilisateur en charge d'un ou plusieurs stades, chaque stade ayant sont référant. Il n'y a qu'un capteur par zone du stade, chaque stade ayant plusieurs zones, un stade possède plusieurs mesures, alors qu'une mesure ne référence qu'un seul stade.

Diagramme d'expérience utilisateur

Le diagramme suivant représente les différents cas pris en charge par la solution applicative :

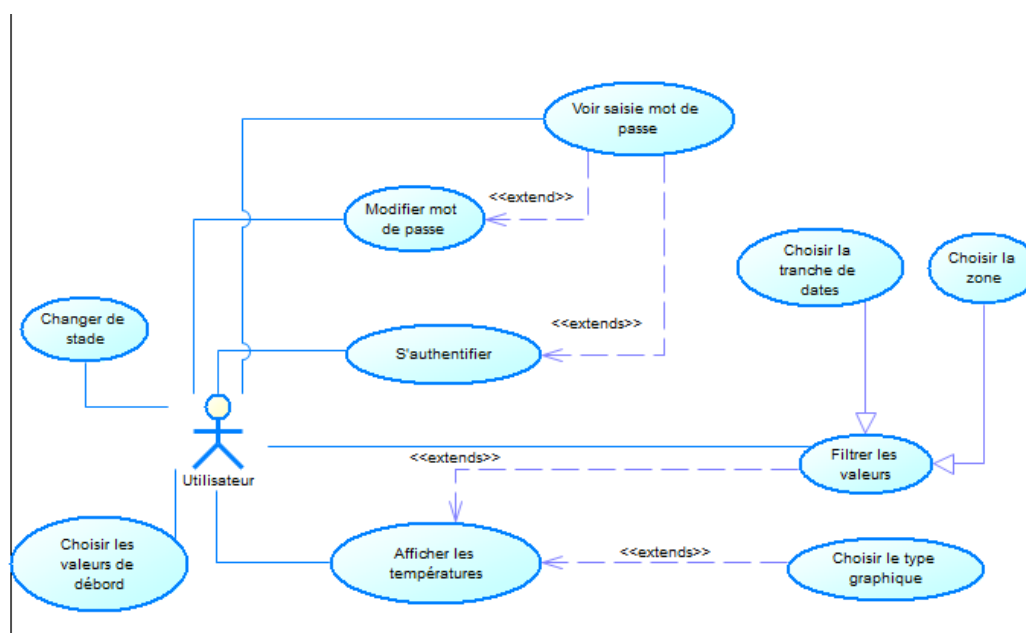


Figure 2 : Diagramme d'expérience utilisateur

On représente les fonctionnalités principales de l'application, soit l'authentification de l'utilisateur, la vérification des mesures effectuées, la gestion multistade ainsi que le filtrage des mesures effectuées.

Diagramme de séquence « boîte noire »

Le diagramme suivant montre le fonctionnement de l'application par l'utilisateur vu comme une boîte noire :

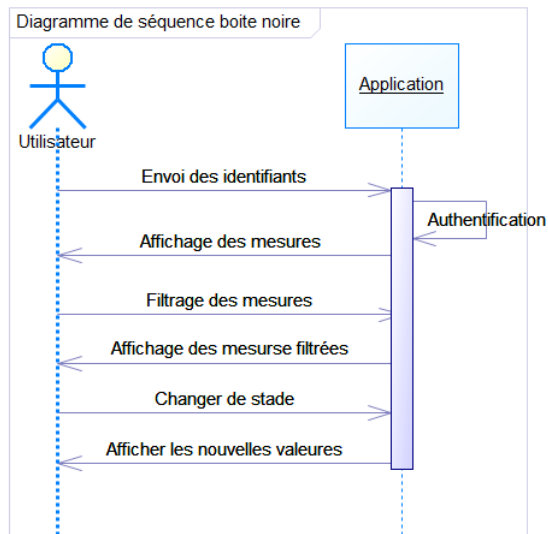


Figure 3 : Diagramme de séquence en boîte noire

Dans ce diagramme, les interactions entre l'utilisateur et l'application sont représentées comme si l'application et son fonctionnement interne nous étaient totalement inconnus.

Il est donc question du schéma de fonctionnement classique et de l'application.

Le second diagramme ci-dessous montre le cas du changement de mot de passe par l'utilisateur :

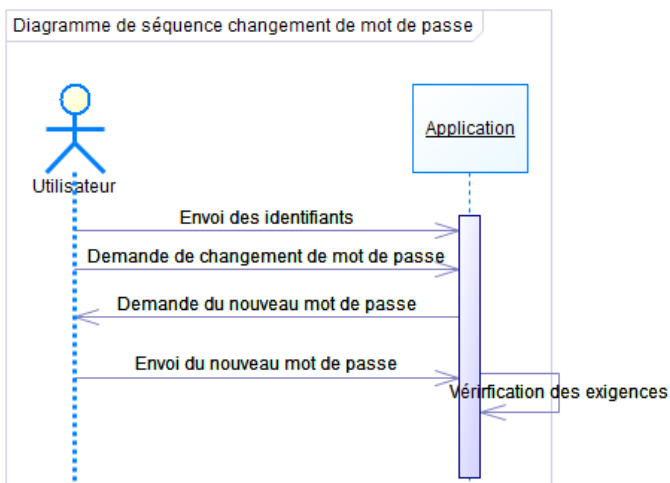


Figure 4 : Diagramme de séquence en boîte noire du changement de mot de passe

Diagramme de séquence

Le diagramme suivant représente le fonctionnement complet de la solution applicative sans prendre en compte les cas d'erreur.

Remarque : Le cas d'utilisation du changement de mot de passe n'est pas représenté ici.

La figure suivante montre une partie du diagramme de séquences énoncé, le fichier complet est disponible dans le répertoire « data » du build, sous le nom « Diagramme_sequence_complet.mo ». ».

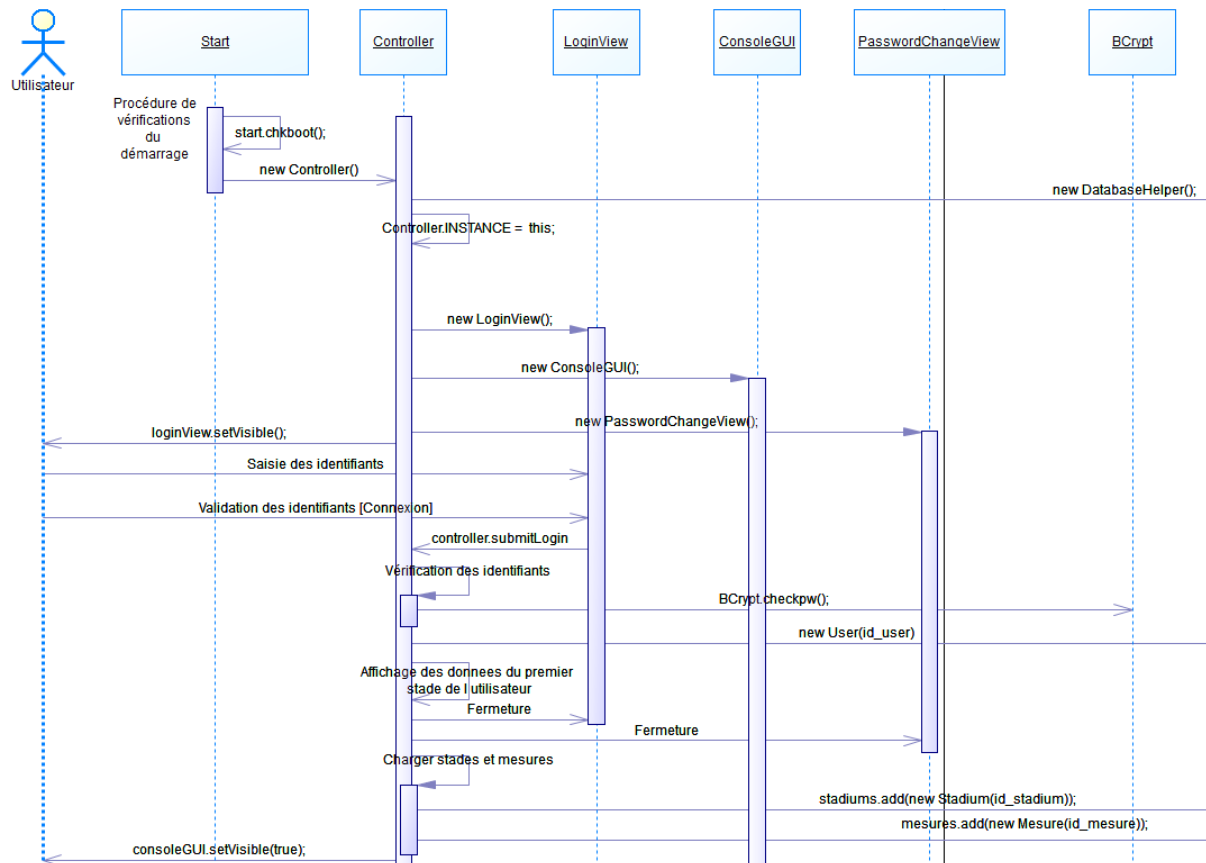


Figure 5 : Aperçu du diagramme de séquence complet de l'application

Il est important de noter que ce diagramme n'est pas représentatif de l'intégralité des interactions au sein de l'application. Celui-ci a été simplifié à des fins de compréhension.

Ce diagramme illustre un cas typique d'utilisation où l'utilisateur s'authentifie, puis vérifie les relevés de température, puis change de stade sélectionné, tout en filtrant les données affichées.

Conception

Architecture

Dans la conception de cette application, nous mettrons en place une architecture MVC¹ (Model View Controller).

Cette architecture sépare les responsabilités des classes en trois parties distinctes :

1. Le contrôleur, il contient la logique de l'application et fait communiquer les autres briques logicielles, soit les vues et les modèles. Il contient également la logique centrale de l'application.
2. La vue, elle gère l'affichage et la synchronisation des données à afficher à l'utilisateur
3. Le modèle, celui-ci s'occupe de la communication avec la base de données, en effet, un modèle est une classe qui représente les enregistrements en base de données et facilite la communication entre le contrôleur et celle-ci. Les modèles contiennent généralement la logique propre aux objets.

L'architecture MVC peut être représentée par le schéma suivant :

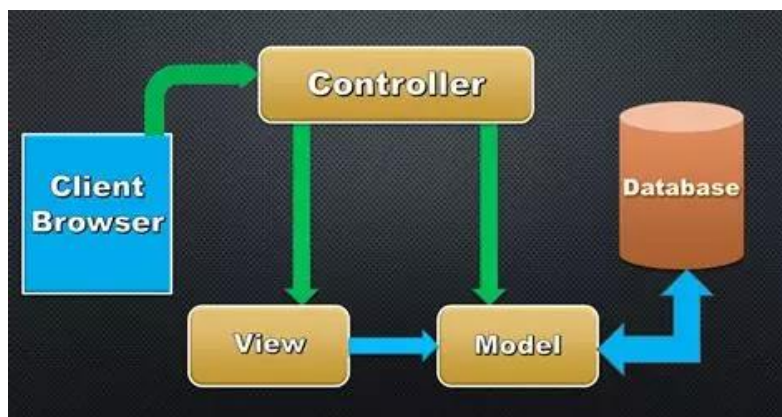


Figure 6 : Architecture MVC

Les classes métier décrits dans l'analyse de conception seront donc des modèles en accord avec l'architecture MVC.

¹ <https://fr.wikipedia.org/wiki/Mod%C3%A8le-vue-contr%C3%B4leur>

Implémentation des classes métier

Dans cette partie du document, nous aborderons l'implémentation des classes métiers.

Ces classes font référence aux modèles du point de vue de l'architecture MVC vue précédemment.

Les classes métier contiennent toute la logique propre aux objets en jeu dans la solution applicative, c'est-à-dire les stades, utilisateurs et mesures.

Remarque : Tous les modèles implémentent une interface nommée « IModel ». Cette interface décrit les méthodes abstraites génériques faisant le lien entre l'application et la base de données, telle que l'**insertion**² et la **mise à jour**³.

Ces méthodes sont ensuite redéfinies dans les classes qui implémentent cette interface. Les méthodes redéfinies sont les suivantes :

```
➤ public void save() ;
```

Cette méthode se charge de la mise à jour des données. Appeler cette méthode a pour effet de mettre à jour l'enregistrement en base de données à l'aide de la requête suivante :

```
st.execute('UPDATE Mesure SET Mesure.num_zone = "' + this.numZone + '"',
Mesure.Date_mesure=' ' + this.horoDate + ', Mesure.Temp=' ' +
this.fahrenheit + ', Mesure.ID_Stadium=' ' + this.IDStadium + «' where
Mesure.ID_Mesure =» + this.id + «;»);
```

Figure 7 : Mise à jour d'une mesure à titre d'illustration

```
➤ public void insert();
```

Cette méthode se charge d'insérer un enregistrement en base de données. Lorsqu'une instance d'un modèle est créée, il est possible d'appeler cette méthode pour insérer un nouvel enregistrement en base de données avec les valeurs des attributs de la classe instanciée.

```
st.execute(«INSERT INTO Mesure (num_zone, Date_mesure, Temp, ID_Stadium)
VALUES ("'+this.numZone+'", "' +format.format(this.horoDate)+'", "' +this.fahren
heit + "', "' + this.IDStadium + "')");
```

Figure 8 : Insertion d'une mesure à titre d'illustration

Remarque : L'intérêt d'utiliser une interface est que si l'on souhaite ajouter une méthode telle que décrite ci-dessus, alors toutes classes qui implémentent cette interface devront redéfinir la méthode. En outre, l'on est sûr que toutes les classes possèdent les mêmes méthodes.

² <https://sql.sh/cours/insert-into>

³ <https://sql.sh/cours/update>

Classe modèle mesure

Accordé à l'analyse de conception, la classe mesure représente les mesures effectuées par les capteurs présents dans les zones d'un stade, on y retrouve alors trois attributs :

- La zone du stade
- La date et l'heure de la mesure
- La valeur de la mesure en degrés Fahrenheit

La figure suivante décrit les attributs définis dans la classe :

```
private String numZone;  
private Date horoDate;  
private float fahrenheit;
```

Figure 9 : Attributs de la classe modèle mesure

La figure suivante décrit le constructeur de cette classe :

```
public Mesure(String pZone, Date pDate, float pFahrenheit) {  
    this.numZone = pZone;  
    this.horoDate = pDate;  
    this.fahrenheit = pFahrenheit;  
}
```

Figure 10 : constructeur de la classe modèle mesure

Ce modèle contient également un getter permettant de renvoyer la valeur de la mesure en degrés Celsius, convertissant la valeur en degrés Fahrenheit :

```
public float getCelsius() {  
    // return (float) (valFahrenheit - 32) / 1.8)  
    return (fahrenheit - 32.0f) / 1.8f;  
}
```

Figure 11 : Définition de la méthode « getCelsius() » de la classe modèle mesure

Classe modèle stade

La classe modèle de stade possède les attributs suivants :

```
private String id;  
private String name;  
private int id_user;
```

Figure 12 : Attributs de la classe stade

Soit l'identifiant du stade, son nom, ainsi que l'identifiant de l'utilisateur en charge du stade.

Remarque : Avec cette solution, il n'est pas possible d'affecter plusieurs utilisateurs à un même stade, limitant les possibilités de gestion des stades.

Le constructeur redéfini de la classe prend en paramètre l'identifiant utilisé en base de données, cela permet de construire la nouvelle instance du modèle en utilisant directement les informations enregistrées en base de données.

Un second constructeur prend en paramètre toutes les valeurs décrites ci-dessus. Tel que le présente la figure ci-dessous :

```
public Stadium(String ID_Stadium, String nom_stade, int id_user) {  
    this.id = ID_Stadium;  
    this.name = nom_stade;  
    this.id_user = id_user;  
}
```

Figure 13 : Second constructeur de la classe stade

En plus des getters et setters pour chacun des attributs, la classe possède deux méthodes permettant respectivement de récupérer **les zones**, ainsi que **les mesures** liées à un stade.

Ainsi, via cette solution, pour un stade sélectionné, il est possible de récupérer toutes les données liées au stade, de les modifier puis de les enregistrer avec les méthodes redéfinies depuis l'interface « IModel » décrite plus tôt.

Classe modèle utilisateur

La classe utilisateur possède les attributs suivants :

```
private int id;
private String name;
private String password;
```

Figure 14 : Attributs de la classe utilisateur

Tels que l'identifiant de l'utilisateur, son nom ainsi que son mot de passe haché.

Remarque : Avec l'implémentation de la bibliothèque **JBCrypt**⁴, le mot de passe de l'utilisateur est haché avec l'algorithme de hachage **BCrypt**⁵. Cela permet d'éviter les risques de vol de mot de passe si les données de la base de données venaient à se faire voler.

Outre les constructeurs et les méthodes redéfinies déjà présentées précédemment, la classe utilisateur possède une méthode **updatePassword** qui permet de mettre à jour le mot de passe de l'utilisateur directement en base de données. La figure ci-dessous montre la méthode **updatePassword** :

```
public boolean updatePassword(String newPassword) {
    String encrypted = BCrypt.hashpw(newPassword, BCrypt.gensalt());
    try {
        Statement st = Controller.INSTANCE.getDB().getCon().
createStatement();

        st.execute(«UPDATE AppUser SET AppUser.Password = """ +
encrypted + """ WHERE AppUser.id_user = ' + this.id + ';'»);

        return true;
    } catch (SQLException e) {
        e.printStackTrace();
        return false;
    }
}
```

Figure 15 : Méthode updatePassword de la classe utilisateur

Cette méthode prend en paramètre le nouveau mot de passe en clair saisi par l'utilisateur.

Le mot de passe est haché avec la bibliothèque **JBCrypt**, et de sa fonction « **hashpw** », puis est mis à jour dans la base de données avec la requête SQL utilisée.

⁴ <https://www.mindrot.org/projects/jBCrypt/>

⁵ <https://fr.wikipedia.org/wiki/BCrypt>

Implémentation des vues

Afin d'afficher les données de façon graphique, le Framework Java **Swing**⁶ a été choisi.

Java **Swing** est un framework de développement d'interfaces graphiques (GUI)⁷ pour les applications Java. Il offre une bibliothèque complète de composants graphiques, tels que des boutons, des fenêtres, des champs de texte, des barres de défilement, etc., permettant aux développeurs de créer des applications avec une interface utilisateur riche et interactive.

Swing offre de nombreuses fonctionnalités avancées, telles que la gestion des événements, la mise en page flexible, la gestion des images, la prise en charge des icônes et des thèmes, ainsi que des fonctionnalités de glisser-déposer. Il est largement utilisé dans le développement d'applications de bureau Java et reste une option solide pour la création d'interfaces graphiques conviviales. Cependant, il convient de noter que Swing coexiste également avec d'autres technologies GUI Java plus récent, telles que **JavaFX**⁸.

⁶ <https://www.javatpoint.com/java-swing>

⁷ https://fr.wikipedia.org/wiki/Interface_graphique

⁸ <https://openjfx.io/>

Vue console

La vue de la console est la vue principale de l'application. Celle-ci est décrite par une maquette Pencil dans le répertoire « data » du build sous le nom « maquette_thermo_green.ep ».

Implémentée sous Swing, la vue ressemble à la figure ci-dessous :

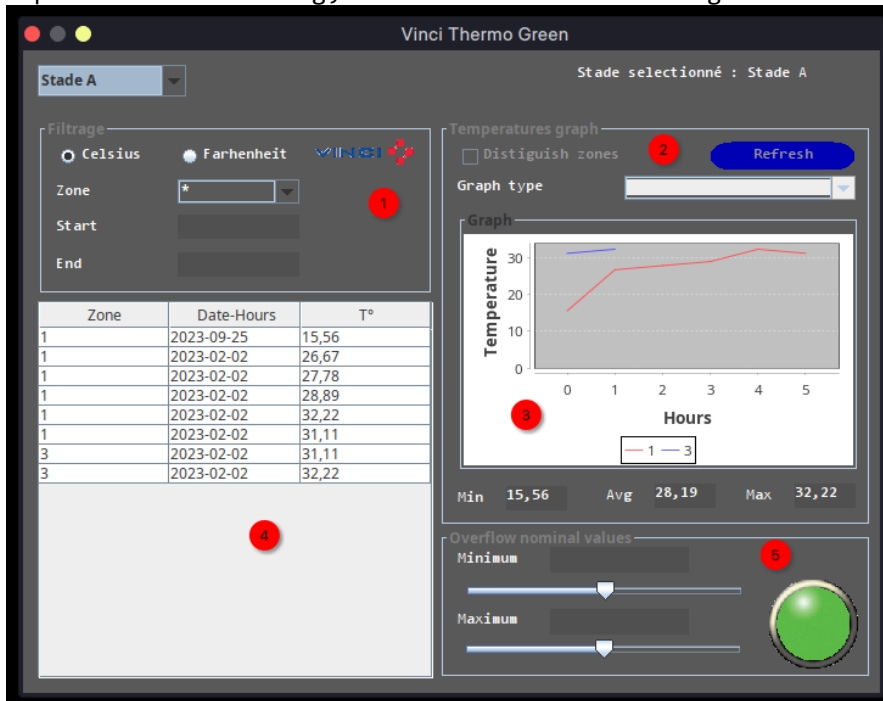


Figure 16 : Implémentation de la vue console sous Java Swing

Cette vue répond aux besoins exprimés dans l'analyse de conception réalisée, tels que l'affichage des relevés de température et le filtrage des données par zones du stade et heure ainsi que le choix du type de graphique.

L'encadré annoté 1 sur la figure permet de filtrer les mesures indiquées dans le tableau déroulant annoté 4 sur la figure.

L'encadré « Temperatures graph » annoté 2 et 3 sur la figure permet de changer de type de graphique et permet de visualiser les relevés de températures sous forme graphique.

L'encadré annoté 5 sur la figure permet de modifier les valeurs de débord. Autrement dit, les limites inférieure et supérieure de température acceptable. Si une température ne correspond pas à ces limites, alors le stade sélectionné sera mis en avertissement.

La figure ci-dessous montre la partie en charge du changement de stade en cours de visualisation :



Figure 17 : Partie permettant de changer de stade en cours de visualisation.

Pour changer de stade sélectionné, il suffit de dérouler la liste, puis de sélectionner le stade voulu.

Vue de connexion

La vue de connexion s'affiche au démarrage de l'application, celle-ci a pour but de récupérer les identifiants de l'utilisateur. La figure ci-dessous montre la vue de connexion :

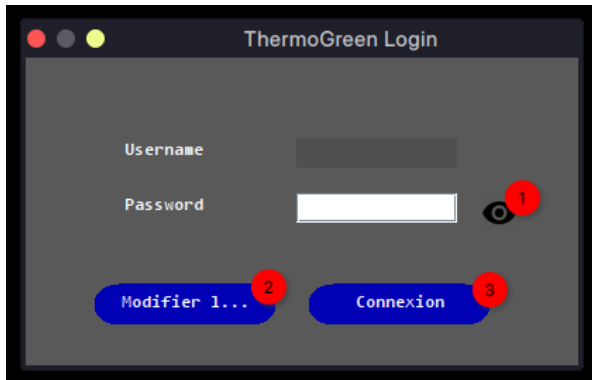


Figure 18 : Vue de connexion

Cette vue ne comporte que deux champs, soit le champ pour le nom d'utilisateur et le champ pour le mot de passe. Si l'utilisateur ne rentre pas d'identifiants, où qu'il se trompe sur les identifiants, un message d'erreur s'affiche, les figures ci-dessous montrent les deux messages d'erreurs possibles :

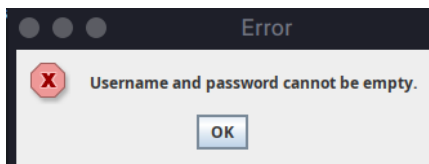


Figure 19 : Message d'erreur si les identifiants sont vides.

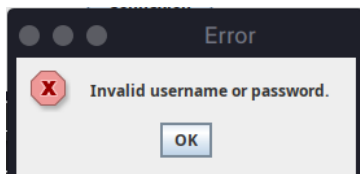


Figure 20 : Message d'erreur si les identifiants sont invalides.

Le bouton annoté 1 permet de visualiser le mot de passe entré et inversement, tel que le montre les figures suivantes :



Figure 21 : Visualisation de la saisie du mot de passe

Vue de changement de mot de passe

La vue changement de mot de passe répond au cas d'utilisation du changement de mot de passe postconnexion. Autrement dit, le mot de passe ne peut être changé que si l'utilisateur se connecte.

La figure ci-dessous représente cette vue :

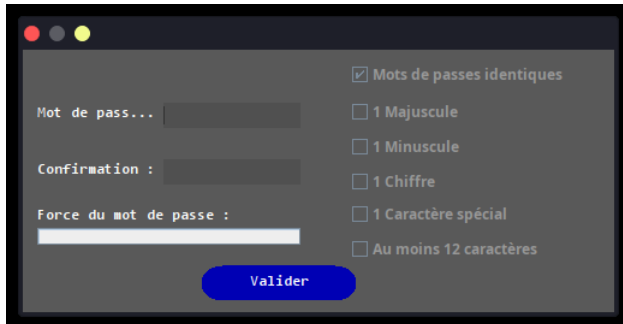


Figure 22 : Vue changement du mot de passe.

Les deux champs de texte permettent de rentrer le mot de passe, ainsi qu'une confirmation du mot de passe. La comparaison entre ces deux champs est effectuée par le contrôleur.

Les critères de sécurité du mot de passe sont vérifiés par le contrôleur également. À chaque frappe de clavier, le contrôleur met à jour les résultats, et les affiche sous forme de cases à cocher, tel que présenté sur la figure ci-dessus.

Lorsque le mot de passe respecte toutes les règles de validation, alors toutes les cases sont cochées, tel que le montre la figure ci-dessous :

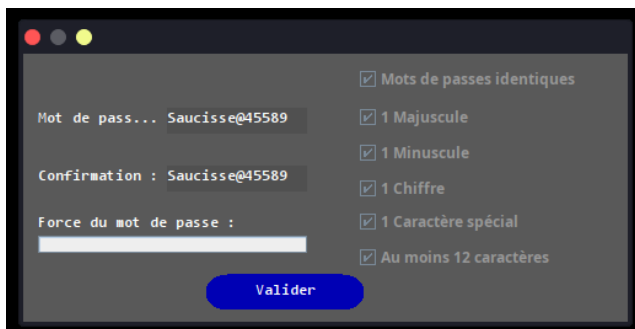


Figure 23 : Visualisation d'un mot de passe valide.

Si le mot de passe ne respecte pas les conditions, le message illustré ci-dessous s'affiche :

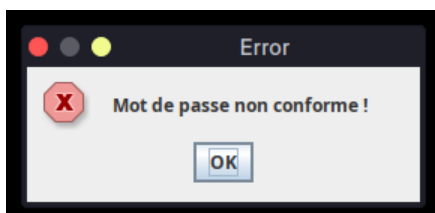


Figure 24 : Erreur de vérification de sécurité des mots de passe.

WindowBuilder

Afin de réaliser facilement les vues et la gestion lors de l'appui sur un élément graphique, l'outil **WindowBuiler** a été utilisé. WindowBuilder est un outil de conception graphique pour le développement d'interfaces utilisateur Java dans l'environnement de développement intégré (IDE) Eclipse. Il simplifie la création d'interfaces utilisateur Swing, SWT (Standard Widget Toolkit), et RCP (Rich Client Platform) en permettant aux développeurs de créer des interfaces utilisateur de manière visuelle, en utilisant des concepts de glisser-déposer et de modification directe.

Les principales caractéristiques de **WindowBuilder** sont les suivantes :

- **Conception visuelle** : WindowBuilder offre une interface graphique intuitive qui permet aux développeurs de créer des interfaces utilisateur en ajoutant des composants graphiques tels que des boutons, des champs de texte, des étiquettes, etc., directement dans l'éditeur visuel.
- **Prise en charge des frameworks** : WindowBuilder prend en charge plusieurs frameworks d'interfaces utilisateur, notamment Swing, SWT, et RCP, ce qui permet aux développeurs de choisir la technologie qui convient le mieux à leur projet.
- **Génération de code automatique** : En travaillant avec l'éditeur visuel de WindowBuilder, le code Java sous-jacent est généré automatiquement, ce qui accélère le processus de développement tout en réduisant les erreurs de codage.
- **Personnalisation avancée** : Les développeurs peuvent personnaliser les composants graphiques directement depuis l'interface visuelle, en ajustant les propriétés et les attributs, et en ajoutant des gestionnaires d'événements.
- **Intégration avec Eclipse** : WindowBuilder est une extension pour Eclipse, ce qui signifie qu'il s'intègre de manière transparente dans l'environnement de développement, offrant une expérience de développement fluide.

Implémentation du contrôleur

Selon l'architecture MVC, le contrôleur fait le lien entre le modèle et la vue. Il écoute la ou les vues et fait changer les modèles en conséquence, et inversement. Il joue le rôle d'aiguillage entre les différentes vues et modèles qui composent l'application.

Dans notre solution applicative, il n'y a qu'un seul contrôleur. Celui-ci est instancié lors du démarrage de l'application, et instancie à son tour les modèles et vues utilisés.

Dans certains cas, il peut être utile d'avoir plusieurs contrôleurs, qui peuvent contrôler plusieurs parties de l'application, cela permet d'avoir une application plus structurée et d'avoir une bonne segmentation des responsabilités de l'application.

Le contrôleur

Le seul et unique contrôleur est instancié lors du démarrage de l'application, comme montré dans la figure ci-dessous :

```
public class Start {

    public static void main(String[] args) {
        try {
            new Start().init();
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }

    public void init() throws ParseException {
        Controller controller = new Controller();
    }
}
```

Figure 25 : Instanciation du contrôleur au démarrage de l'application

Le contrôleur instancie ensuite les modèles et vues en passant en paramètres de ceux-ci l'instance du contrôleur. Cela signifie que les modèles et vues auront un accès direct au contrôleur instancié. La figure ci-dessous montre l'instanciation des modèles et des vues :

```
public Controller() throws ParseException {
    this.locale = Locale.getDefault();
    this.rc = ResourceBundle.getBundle('locale/locale',
    DEFAULT_LOCALE);

    //TODO Replace CSV data by database
    FileUtils.lireCSV(«data/mesures.csv», mesures);

    this.consoleGui = new ConsoleGUI(this);
    this.loginView = new LoginView(this);

    loginView.setVisible(true);

    Measure mesure = new Measure(1);
    System.out.println(mesure.getFahrenheit());
}
```

Figure 26 : Instanciation des vues et du modèle utilisé.

Fonctionnalités du contrôleur

Le contrôleur dispose de plusieurs fonctionnalités permettant aux modèles et vues de communiquer. Autre que les getters et setters, le contrôleur dispose de plusieurs méthodes.

Celles-ci composent les fonctionnalités prises en charge par l'application et sont utilisées par les modèles et les vues, selon le principe de l'architecture MVC.

Vérification des identifiants

La figure ci-dessous montre la méthode « submitLogin » permettant de vérifier les identifiants entrés par l'utilisateur :

```
public void submitLogins(String username, String password) {
    //String fileData = FileUtils.readTxtFile("data/logins.txt");
    //TODO Replace with database access to check credentials

    try {
        DatabaseHelper db = new DatabaseHelper();

        Statement st = db.getCon().createStatement();
        ResultSet result = st.executeQuery('SELECT AppUser.id_user FROM
AppUser WHERE AppUser.username = "" + username + "" and AppUser.password =
"" + password+ "");

        if(!result.next()) {
            JOptionPane.showMessageDialog(loginView.getComponent(0),
                this.rc.getString(«loginViewInvalidCredentials»),
this.rc.getString(«loginViewError»), JOptionPane.ERROR_MESSAGE);
        } else {
            loginView.setVisible(false);
            consoleGui.setVisible(true);
            user = new User(result.getInt('id_user'));
        }

        db.close();
    } catch (ClassNotFoundException | SQLException e) {
        e.printStackTrace();
    }
}
```

Figure 27 : Méthode submitLogin du contrôleur.

Remarque : Cette méthode ne respecte pas totalement les principes de l'architecture MVC dus à la présence de scripts SQL dans le contrôleur. Il est préférable de concentrer les scripts SQL dans les modèles, puisque ceux-ci ont pour but de faciliter la communication entre la base de données et l'application.

Cette méthode permet de vérifier les identifiants de l'utilisateur en se connectant à la base de données puis en vérifiant que l'utilisateur existe et que le mot de passe est correct.

Filtrage des mesures

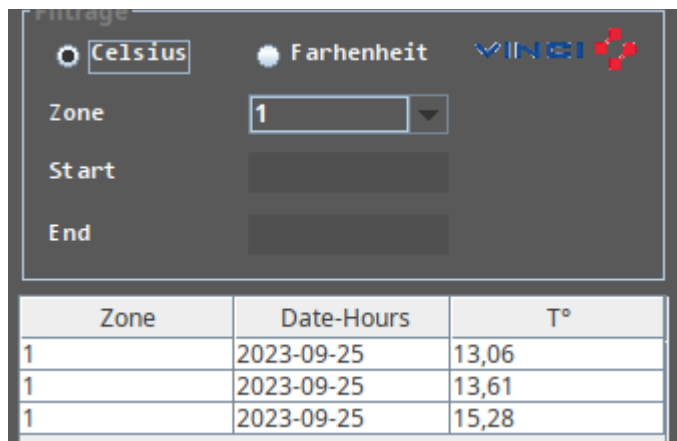
L'autre méthode présente dans le contrôleur permet quant à elle de filtrer les mesures selon les critères de l'utilisateur, la figure ci-dessous montre cette méthode :

```
public ArrayList<Measure> filtrerLesMesures(String laZone) {
    ArrayList<Measure> laSelection = new ArrayList<Measure>();
    for (Measure mesure : mesures) {
        if (laZone.compareTo('*') == 0) {
            laSelection.add(mesure);
        } else {
            if (Integer.parseInt(laZone) == (mesure.getNumZone())) {
                laSelection.add(mesure);
            }
        }
    }
    return laSelection;
}
```

Figure 28 : Méthode « filtrerLesMesures » permettant de filtrer les mesures selon les critères de l'utilisateur

Cette méthode ne filtre les mesures que par rapport à la zone du stade sélectionnée.

La figure ci-dessous présente la vue console, utilisant la fonctionnalité de filtrage :



Zone	Date-Hours	T°
1	2023-09-25	13,06
1	2023-09-25	13,61
1	2023-09-25	15,28

Figure 29 : Vue console utilisant la fonctionnalité de filtrage.

La modification de la valeur sélectionnée dans la liste déroulante fait appel à la méthode présentée ci-dessus dans le contrôleur, qui va filtrer les valeurs, comme montré dans la figure ci-dessus, où le tableau n'affiche que les températures dans la zone 1.

Gestion des valeurs de débord

Le contrôleur s'occupe également de vérifier les valeurs de débord. En effet, la vue console permet de modifier les valeurs supérieures et inférieures de débord grâce à deux sliders, tels que montrés sur la figure ci-dessous :



Figure 30 : Contrôle des valeurs de débord.

Ces valeurs sont envoyées au contrôleur qui va quant à lui les comparer à chaque température en degrés Fahrenheit. Si l'une des températures ne respecte pas la tranche de valeurs définie avec les sliders, la pastille passera au rouge, comme montré sur la figure ci-dessous :

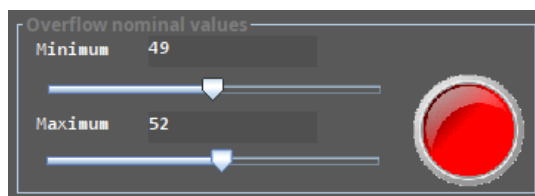


Figure 31 : Températures hors des valeurs définies.

Modification et validation du mot de passe

Si l'utilisateur souhaite modifier son mot de passe, c'est le contrôleur qui va s'occuper de vérifier que le mot de passe respecte bien les conditions définies avec la bibliothèque **Passay**⁹.

Le contrôle vérifie dans un premier temps que le mot de passe correspond aux critères de sécurité, et cela en temps réel. C'est-à-dire qu'à chaque frappe dans les zones de texte, le contrôleur vérifiera les conditions et renverra les résultats à la vue afin de les afficher à l'utilisateur.

Remarque : La procédure précise est décrite dans la partie « [Authentification](#) » du document.

Si le mot de passe est valide, alors le contrôleur procède à la modification du mot de passe en appelant la méthode « updatePassword » de la classe User. Cette méthode permet de mettre à jour le mot de passe dans la base de données.

⁹ Procédure détaillée plus tard dans le document.

Persistence des données

La persistance des données dans une solution applicative est importante pour stocker et conserver les informations de manière durable. Cela permet de garantir la fiabilité, la recherche, l'analyse, l'historique des données, ainsi que l'intégration avec d'autres parties de l'application ou d'autres applications. Cela facilite la prise de décision, la traçabilité, et assure la continuité de l'application.

MySQL

Afin d'assurer la persistance des données, une base MySQL a été mise en place. Le script permettant de créer et d'insérer le jeu d'essai est disponible dans le répertoire « data » du build sous le nom **CREATE_INSERT_VTG.sql**.

La figure ci-dessous présente le script de création des tables avec leurs relations :

```
drop table if exists Mesure;
drop table if exists Stadium;
drop table if exists AppUser;

CREATE TABLE AppUser(
    id_user INT AUTO_INCREMENT,
    Username VARCHAR(64) ,
    Password VARCHAR(64) ,
    CONSTRAINT PK_APPUSER PRIMARY KEY(id_user)
);

CREATE TABLE Stadium(
    ID_Stadium CHAR(5) ,
    nom_stade VARCHAR(64) ,
    id_user INT NOT NULL,
    CONSTRAINT PK_STADIUM PRIMARY KEY(ID_Stadium),
    CONSTRAINT FK_STADIUM_USER FOREIGN KEY(id_user) REFERENCES
AppUser(id_user)
);

CREATE TABLE Mesure(
    ID_Mesure INT AUTO_INCREMENT,
    num_zone INT,
    Date_mesure DATETIME,
    Temp DECIMAL(4,2) ,
    ID_Stadium CHAR(5) NOT NULL,
    CONSTRAINT PK_MESURE PRIMARY KEY(ID_Mesure),
    CONSTRAINT FK_MESURE_STADIUM FOREIGN KEY(ID_Stadium) REFERENCES
Stadium(ID_Stadium)
);
```

Figure 32 : Script de création des tables avec leurs relations.

La structure de la base de données reprend les relations énoncées lors de l'analyse de conception avec les classes métier. Ici, la structure des tables correspond aux attributs présents dans les classes de modèles présentés, afin de permettre de communiquer au mieux avec la base de données.

De cette manière, une instance d'une classe modèle correspond à un enregistrement en base de données.

À la suite de l'implémentation et exécution du script SQL, la figure suivante présente l'aperçu de la structure de la base de données :

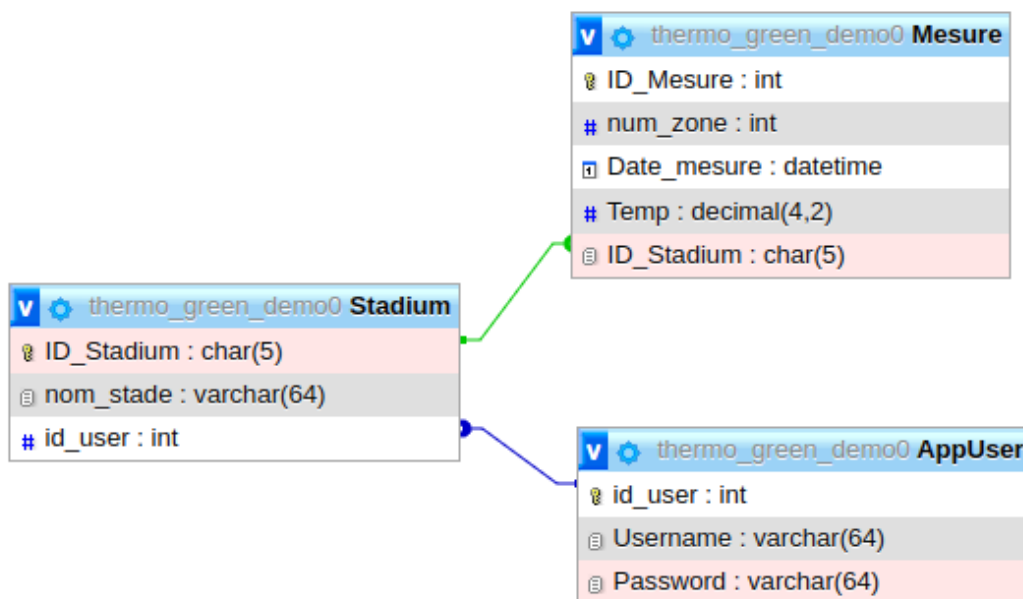


Figure 33 : Aperçu de la structure de la base de données.

Connexion avec la base de données

Après la mise en place de la base de données, la dernière étape est de mettre en place la connexion entre l'application et le serveur de base de données. Pour réaliser cela, l'on a utilisé le connecteur JDBC.

Le connecteur **JDBC**¹⁰ (Java Database Connectivity) est une API de programmation Java permettant d'établir une connexion entre une application Java et une base de données. Il offre un ensemble de classes et de méthodes pour exécuter des requêtes SQL, récupérer et mettre à jour des données dans la base de données.

Le connecteur JDBC est essentiel pour interagir avec des bases de données relationnelles, telles que MySQL, Oracle, PostgreSQL, etc., à partir d'une application Java. Il facilite la gestion des connexions, des requêtes et des transactions, permettant ainsi aux développeurs de créer des applications robustes et performantes qui manipulent des données stockées dans des bases de données.

L'implémentation du connecteur est réalisée dans la classe « DatabaseHelper », instanciée par le contrôleur durant la phase d'utilisation du programme.

¹⁰ https://fr.wikipedia.org/wiki/Java_Database_Connectivity

La classe DatabaseHelper

La classe **DatabaseHelper** sert à établir une connexion à une base de données en utilisant JDBC (Java Database Connectivity). Voici comment elle fonctionne étape par étape :

- Les variables **url**, **username** et **password** sont initialisées pour stocker les informations de connexion à la base de données.

```
private String url = "";
private String username = "";
private String password = "";
```

Figure 34 : Attributs de la classe DatabaseHelper

- Dans le constructeur de la classe, la classe de pilote JDBC pour MySQL est chargée à l'aide de **Class.forName(« com.mysql.cj.jdbc.Driver »)**. Cela enregistre le pilote JDBC dans l'application.
- Un fichier de configuration de la base de données est lu à partir de l'emplacement spécifié dans **Config.DBENVFILEPATH**¹¹. Les informations de connexion (URL, nom d'utilisateur et mot de passe) sont extraites du fichier.
- Une connexion à la base de données est établie en utilisant les informations récupérées à partir du fichier de configuration. Cela se fait avec **DriverManager.getConnection(url, username, password)**.
- La méthode **getCon()** permet de récupérer la connexion établie pour être utilisée dans d'autres parties de l'application.
- La méthode **close()** permet de fermer la connexion à la base de données lorsque vous avez terminé de l'utiliser, afin de libérer les ressources.

La classe **DatabaseHelper** charge le pilote JDBC, récupère les informations de connexion à partir d'un fichier de configuration, établit une connexion à la base de données et fournit des méthodes pour obtenir la connexion et la fermer correctement.

Cela facilite l'accès et la gestion de la base de données dans une application Java.

¹¹ Classe de configuration contenant des variables statiques finales, utilisées tout au long de l'exécution de l'application.

Données de connexion

Stocker les informations de connexion à une base de données dans un fichier offre des avantages clés.

Cela sépare les données de configuration de la logique de l'application, renforce la sécurité, simplifie la maintenance, améliore la portabilité et la flexibilité, tout en permettant un partage plus efficace des paramètres de connexion entre les membres de l'équipe. Cela rend l'application plus modulaire et adaptable, réduisant les risques de fuite de données et simplifiant la gestion des configurations.

Ce fichier nommé « **db.env** », situé dans le répertoire « **data** » stocke les informations suivantes :

Driver	Jdbc
SGBD	MySQL
Adresse IP	192.168.122.245 ¹²
Port	3306
Base de donnée	Thermo_green_demo0
Utilisateur	_gateway
Mot de passe	XXXXX

Ainsi, le fichier est structuré comme montré ci-dessous :

```
jdbc:mysql://192.168.122.245:3306/thermo_green_demo0
_gateway
XXXXX
```

Figure 35 : Fichier contenant les informations de connexion à la base de données.

La première ligne reprend le driver, le SGBD, l'adresse IP du serveur, le port ainsi que le nom de la base de données ciblée.

La seconde ligne est le nom d'utilisateur.

La dernière ligne est le mot de passe.

L'ordre de ces lignes est très important puisque la classe « DatabaseHelper » va lire ce fichier en suivant l'ordre présenté. Ainsi, si le fichier n'est pas structuré tel que présenté, la connexion à la base de données échouera.

Remarque : S'agissant d'un prototype, il ne s'agit pas de la meilleure structure pour un tel fichier.

Lorsque la classe « DatabaseHelper » est instanciée, une nouvelle connexion à la base est créée à l'aide du fichier présenté ci-dessous, le constructeur de la classe utilise une instance de `BufferedReader` afin de lire le fichier ligne par ligne et de concaténer les chaînes de caractères afin de constituer la chaîne de connexion à la base de données.

¹² Adresse IP de la machine virtuelle utilisé à des fins de tests.

La figure ci-dessous présente cette portion du code :

```
reader = new BufferedReader(new FileReader(Config.DBENVFILEPATH));
String line = reader.readLine();

while (line != null) {
    cdx += line + ';' ;
    line = reader.readLine();
}
reader.close();
```

Figure 36 : Lecture du fichier et création de la chaîne de connexion à la base de données.

Enfin, la chaîne créée est divisée pour récupérer l'URL, l'identifiant et le mot de passe, utilisé par la fonction « **getConnection** » de la classe **DriverManager**, comme montré dans la figure ci-dessous :

```
String[] arrc = cdx.split(';');

url = arrc[0];
username = arrc[1];
password = arrc[2];

this.con = DriverManager.getConnection(url, username, password);
```

Figure 37 : Séparation de la chaîne et ouverture de la connexion.

L'instance de la connexion est ensuite stockée dans la variable « con » de la classe **DatabaseHelper**, et utilisable par le contrôleur et les modèles pour accéder à la base de données.

Authentification

Afin de garantir la sécurité et la confidentialité des données traitées par Vinci. Il a été question lors de l'analyse de conception du cas d'utilisation de l'authentification des utilisateurs.

En effet, pour accéder aux données et changer son mot de passe, les utilisateurs sont dans l'obligation de rentrer un mot de passe, ainsi qu'un identifiant afin de s'authentifier après de l'application.

Ce cas d'utilisation est traité de manière graphique avec la vue de connexion présentée lors du détail de la conception des vues de l'application.



Figure 38 : Connexion d'un utilisateur

Or, l'authentification est gérée du côté du client, par l'application elle-même. Celle-ci vise à, dans un premier temps, vérifier l'existence de l'utilisateur dans la base de données. Si l'utilisateur n'existe pas, ce message apparaît alors :

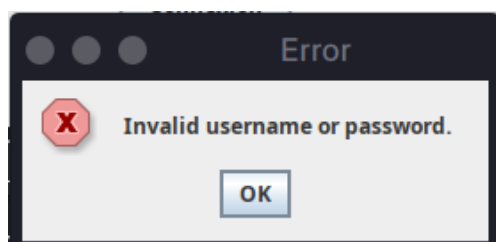


Figure 39 : Erreur, refus de connexion

Si l'utilisateur existe, mais que le mot de passe est incorrect, alors le même message apparaît. Cela permet de ne pas donner d'indices dans le cas d'une attaque par force brute¹³.

Bien que l'utilisateur rentre son mot de passe en clair, celui-ci est haché afin de le rendre « invisible » par quiconque accèderait à la base de données.

¹³ https://fr.wikipedia.org/wiki/Attaque_par_force_brute

Hash du mot de passe

Les **hashs** sont des empreintes numériques générées à partir de données, offrant des avantages importants. Ils vérifient l'intégrité des données, sécurisent les mots de passe, simplifient les comparaisons, renforcent la sécurité des communications et réduisent la complexité des données. Les **hashs** sont essentiels dans la protection des données, la cryptographie et la gestion efficace de l'information.

Un hash est donc une empreinte qui représente de données, il n'est donc pas possible de « dé-hacher » une empreinte.

Pour vérifier la validité du mot de passe, il suffit alors de hacher l'entrée de l'utilisateur, puis de comparer les valeurs avec celle présente en base de données. L'on vérifie alors le mot de passe.

Cela permet donc de ne pas traiter le mot de passe en clair dans l'application, offrant une couche de sécurité supplémentaire.

Implémentation

L'implémentation du hash se fait avec la bibliothèque « **JBcrypt**¹⁴ », qui permet d'implémenter l'algorithme de hash « **BCrypt**¹⁵ » dans un programme Java.

JBcrypt est une classe contenant des fonctions statiques, pouvant donc être appelée n'importe où dans le programme de la manière suivante :

```
BCrypt.nom(params) ;
```

Cette classe contient diverses fonctions, dont deux principales permettant respectivement de hacher et comparer des chaînes de caractères.

La fonction suivante permet de hacher un mot de passe :

```
BCrypt.hashpw («motdepasse 1234») ;
```

Cette fonction renvoie un hash, sous cette forme :

```
$2y$10$8U7Hz8LuHvkqMgHNXlbuwOWc0J7VcI40lJd7DedNhqrfziUjr5P3G
```

Figure 40 : Exemple de hash avec l'algorithme BCrypt.

La seconde fonction permet de comparer un mot de passe en clair avec un hash :

```
BCrypt.checkpw («motdepasse 1234»,  
«$2y$10$8U7Hz8LuHvkqMgHNXlbuwOWc0J7VcI40lJd7DedNhqrfziUjr5P3G») ;
```

Cette fonction renvoie un booléen vrai si la comparaison est validée, ou fausse sinon.

Ces deux fonctions sont appelées lorsque l'on a besoin de vérifier un mot de passe, ou bien dans ajouter un à la base de données lors de la création d'un nouvel utilisateur ou lors du changement d'un mot de passe par exemple.

¹⁴ <https://www.mindrot.org/projects/jBCrypt/>

¹⁵ <https://fr.wikipedia.org/wiki/BCrypt>

Force et prérequis d'un mot de passe

Afin de garantir la sécurité de l'application, il est nécessaire d'imposer certaines conditions lors du changement d'un mot de passe.

Pour cela, on utilise la bibliothèque « Passay ».

La bibliothèque Passay est une bibliothèque open source conçu pour gérer la validation et la génération de mots de passe sécurisés. Elle offre des fonctionnalités pour créer, valider et améliorer la qualité des mots de passe, ce qui contribue à renforcer la sécurité des systèmes d'authentification.

Passay permet de définir des règles de complexité pour les mots de passe, comme la longueur minimale, l'utilisation de caractères spéciaux, de chiffres, de lettres majuscules, etc.

En outre, la bibliothèque offre des mécanismes de personnalisation pour s'adapter aux besoins spécifiques des applications et garantir que les mots de passe utilisés dans un système sont robustes et difficiles à deviner, renforçant ainsi la sécurité des comptes utilisateur.

Passay est implémenté pour répondre au cas d'utilisation du changement de mot de passe, la figure suivante montre la vue permettant de changer le mot de passe :

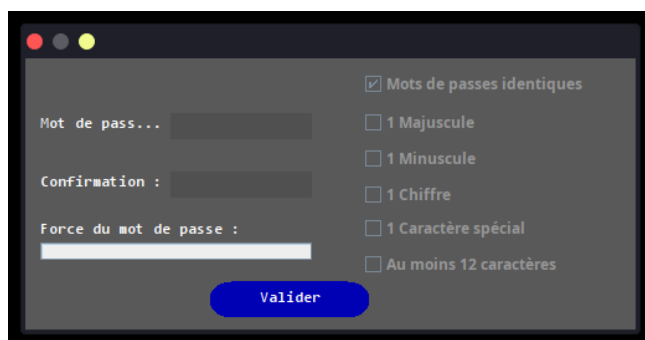


Figure 41 : Vue de changement du mot de passe.

La bibliothèque Passay est utilisée pour vérifier la correspondance du nouveau mot de passe aux critères listés à gauche de la fenêtre, représenté par des boîtes à cocher.

Ainsi, pour valider le nouveau mot de passe, toutes les cases doivent être cochées, comme présenté dans la figure ci-dessous :

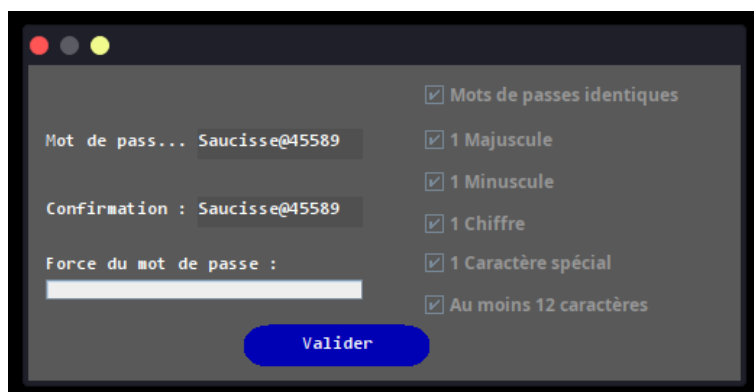


Figure 42 : Validation du nouveau mot de passe avec Passay.

Validation du mot de passe

Toute la logique de validation du mot de passe se trouve dans le contrôleur de l'application.

Les conditions de validité du mot de passe sont définies dans le constructeur du contrôleur, comme montré ci-dessous :

```
passwordValidator = new PasswordValidator(  
    new LengthRule(Config.MIN_PASSWORD_LENGTH,  
Config.MAX_PASSWORD_LENGTH),  
    new CharacterRule(EnglishCharacterData.LowerCase, 1),  
    new CharacterRule(EnglishCharacterData.UpperCase, 1),  
    new CharacterRule(EnglishCharacterData.Digit, 1),  
    new CharacterRule(EnglishCharacterData.Special, 1),  
    new IllegalSequenceRule(EnglishSequenceData.Alphabetical, 4, false),  
    new IllegalSequenceRule(EnglishSequenceData.Numerical, 4, false),  
    new IllegalSequenceRule(EnglishSequenceData.USQwerty, 4, false),  
    new WhitespaceRule());
```

Figure 43 : Définition des règles de validation du mot de passe.

Une fois que l'utilisateur entre le mot de passe, à chaque saisie, le mot de passe entré dans le premier champ intitulé « mot de passe : » dans la vue est vérifié par Passay.

Dans le contrôleur, la méthode « changePassword » est appelée en lui passant en paramètres le nouveau mot de passe, celui-ci est vérifié en fonction des conditions de validation énoncées dans le constructeur de la classe contrôleur, indiqué sur la figure précédente.

Afin de personnaliser les messages d'erreur si le mot de passe est invalide, l'on utilise une méthode callback appelée à chaque saisie de l'utilisateur et redéfinie pour chaque condition de validations.

Autrement dit, à chaque saisie, le mot de passe est vérifié. Pour chaque condition de validation, un booléen est renvoyé par la méthode callback correspondant aux statuts de la condition, si celle-ci est respectée ou non par le mot de passe.

Enfin, pour chaque condition, on coche ou non la case correspondante et ainsi montrer à l'utilisateur le statut de son mot de passe en lui indiquant les règles à respecter pour avoir un mot de passe correct.

Internationalisation

Dans l'optique de rendre accessible cette solution logicielle à des salariés non francophones, Vinci souhaite ajouter la possibilité de changer la langue d'affichage du logiciel.

L'intérêt est donc de mettre en avant l'accessibilité du logiciel pour tous les salariés susceptibles de travailler avec celui-ci.

Implémentation

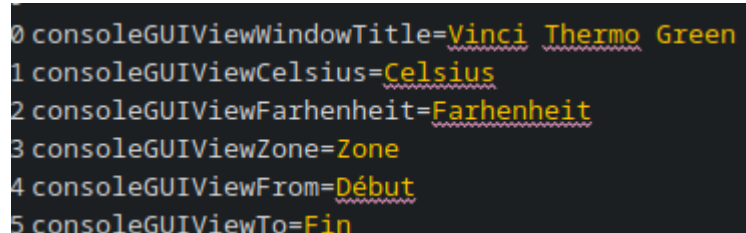
L'implémentation de l'internationalisation se fait par le biais de la classe « `ResourceBundle` » du package `Java.utils` du JDK.

La classe « `ResourceBundle` ¹⁶ » du JDK est utilisée en Java pour gérer des ressources localisées, telles que des textes traduits, des fichiers de configuration, et d'autres éléments en fonction de la langue ou de la région de l'utilisateur.

Elle facilite la localisation des applications, la gestion des paramètres régionaux, la centralisation des ressources, et permet de fournir des interfaces utilisateur adaptées aux préférences linguistiques et régionales des utilisateurs.

Cette classe permet de créer un couple de clés-valeur entre un identifiant et un texte traduit. Ces couples se situent dans un fichier de traduction nommé « `locale_[Langue].properties` ».

La figure ci-dessous présente une partie du fichier de traduction française :



```
0 consoleGUIViewWindowTitle=Vinci Thermo Green
1 consoleGUIViewCelsius=Celsius
2 consoleGUIViewFarhenheit=Farhenheit
3 consoleGUIViewZone=Zone
4 consoleGUIViewFrom=Début
5 consoleGUIViewTo=Fin
```

Figure 44 : Partie du fichier de traduction française.

Ensuite, on change les données de traduction, stockée dans une variable du type « `ResourceBundle` ». On utilise la fonction statique « `getBundle` », en passant en paramètre, le chemin relatif au projet et le nom de la ressource.

De cette manière, il est possible de lier une langue à un fichier précis.

Enfin, il ne reste plus qu'à récupérer la valeur souhaitée en spécifiant la clé, qui retourne la valeur associée, en fonction de la langue choisie.

Par exemple, la clé « `consoleGUIViewFrom` » équivaudra à « `Début` » en Français et « `From` » en Anglais. L'avantage est que la clé reste la même, qui importe la langue.

¹⁶

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/ResourceBundle.html>

On récupère une valeur à l'aide de la méthode « `getString()` » de l'instance de la classe « **ResourceBundle** », instanciée au préalable en spécifiant la langue désirée.

Gestion des erreurs

La gestion des erreurs est une part importante lorsqu'il s'agit de garantir la fiabilité et l'accessibilité d'une application.

Afin de gérer les principaux cas d'erreurs, nous avons implémenté une phase de pré démarrage afin de vérifier certains paramètres qui pourraient partiellement ou totalement empêcher le fonctionnement normal de l'application.

Implémentation

Cette vérification est implémentée dans la classe de démarrage « Start ». Cette classe s'occupe d'instancier le contrôleur pour donner suite au démarrage de l'application.

Or, avant de procéder à l'instanciation du contrôleur, ces vérifications du démarrage sont effectuées. La méthode « **chkBoot** » de la classe Start est appelée.

Toutes les erreurs possibles durant les vérifications sont énumérées dans une énumération ¹⁷ nommée « **EError** » (**Enumeration Error**), afin de formaliser les différentes erreurs.

La figure ci-dessous présente cette énumération :

```
public enum EError {  
  
    NO_ERROR("", 0),  
    MISSING_INVALID_CONFIG("Missing or invalid configuration", -1),  
    UNREACHABLE_DATABASE("Database connexion error, please check your database connexion informations", -2);  
  
    private String errMsg;  
    private int errCode;  
  
    private EError(String errMsg, int errCode) {  
        this.errMsg = errMsg;  
        this.errCode = errCode;  
    }  
  
    public String getErrMsg() {  
        return errMsg;  
    }  
  
    public int getErrCode() {  
        return errCode;  
    }  
}
```

Figure 45 : Énumération EError

La méthode « **chkBoot** » vérifie certaines conditions, si l'une d'entre elles échouent, alors la méthode renvoie une variable de type « **EError** » contenant les informations en rapport avec l'erreur.

Si toutes les vérifications passent, alors la méthode renvoie une valeur **EError.NO_ERROR**, qui correspond à l'absence d'erreur, l'application peut donc démarrer correctement.

¹⁷ https://www.w3schools.com/java/java_enums.asp

Exemples de cas d'erreur

Le premier cas d'erreur traité est la présence ou non du fichier contenant les informations de connexion à la base de données.

Pour vérifier cela, l'on crée une variable de type « File » avec le chemin relatif du fichier de configuration, puis l'on vérifie sa présence avec la condition « file.exists() ».

Si le fichier n'existe pas, une erreur « MISSING_INVALID_CONFIG » est renvoyée, tel que le montre la figure suivante :

```
File file = new File(Config.DBENVFILEPATH);
if(!file.exists()) return EError.MISSING_INVALID_CONFIG;
```

Figure 46 : Traitement du cas d'erreur numéro 1

Le second cas d'erreur est la connexion à la base de données. Si le fichier est présent, cette seconde vérification permet de s'assurer que la connexion à la base de données est possible.

Si ce n'est pas le cas, cela suggère une panne du côté du serveur ou des informations de connexion erronées.

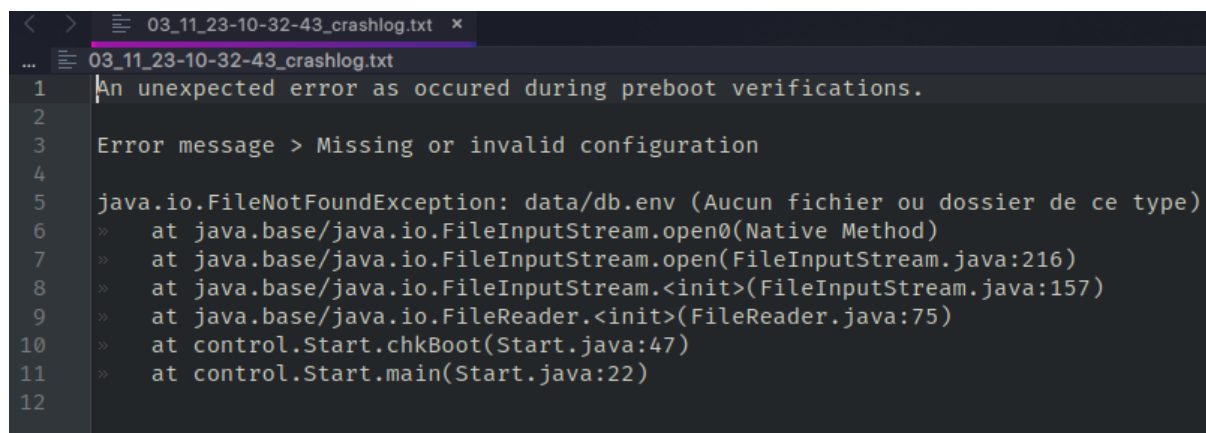
Afin de vérifier la connexion à la base, une instance de la classe « DatabaseHelper » est créée. Si une exception est levée lors de l'instanciation, cela signifie que la connexion ne s'est pas produite normalement. Une erreur « UNREACHABLE_DATABASE » est levée, est l'application s'arrête.

Journalisation

Si une erreur se produit au démarrage, un fichier de journalisation contenant la **stacktrace**¹⁸ de l'erreur est générée et placée dans le même répertoire où se situe l'exécutable .JAR de l'application.

La méthode « writeCrashLog », qui prend en paramètre l'erreur et l'exception à l'origine du crash produit un fichier où se trouve l'exécutable JAR.

La figure ci-dessous présente un fichier de journalisation de crash :



```

1 An unexpected error as occured during preboot verifications.
2
3 Error message > Missing or invalid configuration
4
5 java.io.FileNotFoundException: data/db.env (Aucun fichier ou dossier de ce type)
6   at java.base/java.io.FileInputStream.open0(Native Method)
7   at java.base/java.io.FileInputStream.open(FileInputStream.java:216)
8   at java.base/java.io.FileInputStream.<init>(FileInputStream.java:157)
9   at java.base/java.io.FileReader.<init>(FileReader.java:75)
10  at control.Start.chkBoot(Start.java:47)
11  at control.Start.main(Start.java:22)
12
```

Figure 47 : Journalisation d'un crash de l'application.

¹⁸ https://fr.wikipedia.org/wiki/Trace_d%27appels

Gestion des messages d'alerte

Dans le cas où une ou plusieurs températures seraient en dehors des limites imposées, le personnel en charge du stade doit être en mesure de notifier le personnel de maintenance par le biais d'une notification par SMS.

Pour répondre à ce besoin, nous utilisons l'API **Vonage**, qui propose une palette de fonctionnalités, y compris l'envoi de SMS à un numéro de téléphone.

API Vonage

L'API **Vonage**¹⁹ est une plateforme de communication en temps réel qui fournit des services de messagerie, d'appels vocaux et vidéo, ainsi que des fonctionnalités de vérification des numéros de téléphone.

Elle permet aux développeurs d'intégrer facilement des fonctionnalités de communication dans leurs applications, que ce soit pour envoyer des SMS, effectuer des appels vocaux ou vidéo, ou mettre en place des fonctionnalités avancées telles que la vérification à deux facteurs.

En utilisant l'API **Vonage**, il est possible de créer des expériences utilisateur riches et interactives, favorisant la connectivité dans divers contextes d'application.

L'utilisation de l'API nécessite la création d'un compte utilisateur. En effet, cet outil n'est pas gratuit et un compte est nécessaire pour créditer l'utilisation de l'API.

La création du compte permet d'obtenir dans un premier temps deux euros de crédits gratuits pour tester les fonctionnalités de l'API, tel que le montre la figure suivante du tableau de bords :

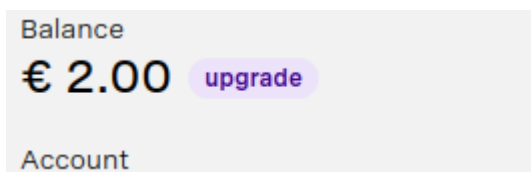


Figure 48 : Crédits gratuits à la création d'un compte.

Dans un second temps, il nous sera possible de récupérer les clés permettant de communiquer avec l'API. Il s'agit d'une paire de clés permettant s'authentifier sur le serveur. La figure ci-dessous montre les clés disponibles sur le tableau de bord :

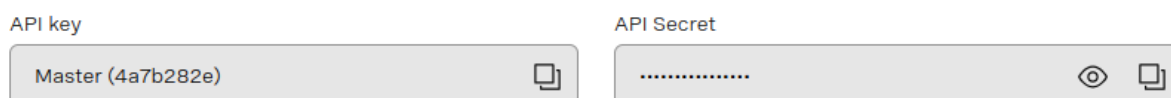


Figure 49 : Paires de clés nécessaires pour communiquer avec l'API.

¹⁹ <https://www.vonage.com/communications-apis/sms/>

Client Java Vonage

Pour permettre aux développeurs d'utiliser l'API **Vonage** par le biais d'un programme en Java, il est possible d'utiliser une bibliothèque cliente Vonage écrite en Java afin d'implémenter les outils proposés par l'API directement dans un projet Java.

Pour récupérer ce client, il suffit de cloner le dépôt Github donné ci-dessous :

<https://github.com/Vonage/vonage-java-sdk.git>

Puis, après avoir ouvert un répertoire dans le dossier créé à la suite du clonage, taper la commande « ./gradlew build ».

Cette commande va rassembler les ressources utilisées par cette bibliothèque, puis compiler les sources du projet pour en produire un fichier .jar.

Ces fichiers sont disponibles dans le sous répertoire « build » généré par **Gradle**²⁰. Il suffit enfin d'ajouter la bibliothèque dans le classpath du projet.

Implémentation

Les fonctionnalités de l'API Vonage sont implémentées dans une classe nommée « **SMSSender** ».

Cette classe instancie un objet « **VonageClient** » qui est en somme l'interface de contrôle entre le programme et l'API. Pour instancier cette classe, il sera nécessaire de fournir la paire de clés fournies sur le tableau de bord.

La figure ci-dessous montre l'instanciation de la classe **VonageClient** :

```
vonageClient = VonageClient.builder()
    .apiKey(controller.getProperties().getProperties().getProperty("api.api_key"))
    .apiSecret(controller.getProperties().getProperties().getProperty("api.api_secret")).build();
```

Figure 50 : Instanciation de la classe **VonageClient**

Ces clés sont stockées dans un fichier « **vonage_api.properties** » et accédées via la classe « **java.util.Properties** » du JDK.

Enfin, la figure ci-dessous montre l'implémentation de l'envoi d'un SMS utilisant le client Vonage :

```
public void sendMessage(String message) {
    TextMessage textMessage = new TextMessage("Vinci Thermo Green",
        controller.getSelectedStadium().getContactNumber(), message.isEmpty() ? "Default Message" : message);

    SmsSubmissionResponse response = vonageClient.getSmsClient().submitMessage(textMessage);

    if (response.getMessages().get(0).getStatus() == MessageStatus.OK) {
        System.out.println("Message sent successfully.");
    } else {
        System.out.println("Message failed with error: " + response.getMessages().get(0).getErrorText());
    }
}
```

Figure 51 : Implémentation de l'envoi d'un SMS par le client Vonage.

²⁰ <https://gradle.org/>

Le numéro de téléphone du correspondant est récupéré depuis le stade sélectionné par le contrôleur.

Le SMS est construit dans un objet « TextMessage » et est composé du numéro de téléphone ainsi que d'un message.

Ensuite, le message est envoyé en utilisant la méthode « submitMessage() » de la classe VonageClient en lui passant en paramètre l'objet TextMessage.

La figure ci-dessous montre le récapitulatif des messages envoyés sur le tableau de bord associé au compte utilisateur de l'API :

Time range : Last hour
Direction : Outbound

Search
Download CSV
Clear selectio

1 results • 4a7b282e • outbound

Filter by

☐
Status
Country
Latency
Network
Message body
Export selected to CSV

<input type="checkbox"/>	Message ID & API key	From ⓘ	To	Status	Latency ⓘ	Network ⓘ	Body
<input type="checkbox"/>	8c419ef4-c120-4179-ba9e-5c885a4415c9 4a7b282e API key	Vinci Thermo Green	33619575215 FR	accepted		SFR 20810	test message[FREE SMS DEMO, TEST MESSAGE]

Figure 52 : Récapitulatif des messages envoyés par l'API.

On y retrouve le numéro de téléphone du destinataire ainsi que le contenu du message et le statut du SMS.

Documentation et déploiement

La documentation est la première partie concernant le déploiement d'un projet. En effet, la documentation consiste à expliquer en détail le fonctionnement du code et ses spécificités sous formes de commentaires directement mis dans le code.

Documentation

Dans le cas de ce projet, la documentation est représentée sous la forme présentée par la figure suivante :

```
/**
 * <p>
 * Change user password, asks for authentication first.
 * </p>
 *
 * @param password - New password of the user who needs to change his password
 * @param confirm - Confirmation of the previous password
 *
 * @author Thomas PRADEAU
 * @version 3.0.0
 */
public void changePassword(String password, String confirm) {
```

Figure 53 : Forme de la documentation sous Java

Il est possible de documenter des classes, fonction, méthode et mêmes les attributs d'une classe. Tous ces commentaires sont ensuite rassemblés dans un document appelé « Javadoc ».

La Javadoc est un ensemble de documents HTML qui regroupent toute la documentation écrite sur les divers éléments du code. Cela permet de faciliter la maintenance et l'évolutivité d'un projet, par exemple dans le cas d'un changement d'équipe de développeurs.

L'écriture de la documentation peut être fastidieuse, c'est pour cela qu'il est recommandé de l'écrire au fur et à mesure de la rédaction du code. Et ce, afin d'éviter d'oublier de documenter certaines parties du code par exemple.

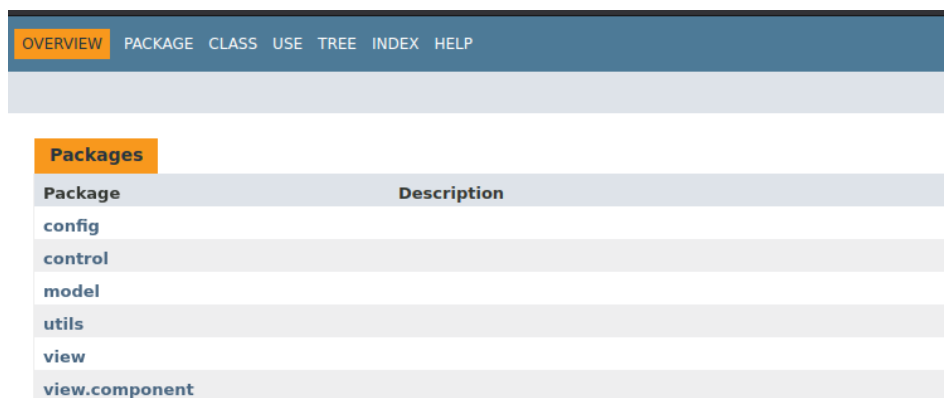
Générer la Javadoc

Ce procédé varie d'un IDE à un autre. Sous Eclipse par exemple, on peut générer la Javadoc en allant sous « Exporter » puis « Javadoc ».

On sélectionne ensuite les classes concernées, puis Eclipse va lire tous les fichiers pour générer les documents HTML.

Remarque : Si des éléments d'une ou plusieurs classes ne sont pas documentés correctement, Eclipse renverra des avertissements et erreurs, cela ne signifie pas que la génération ne s'est pas produite, mais que celle-ci peut ne pas être complète selon les critères d'Eclipse.

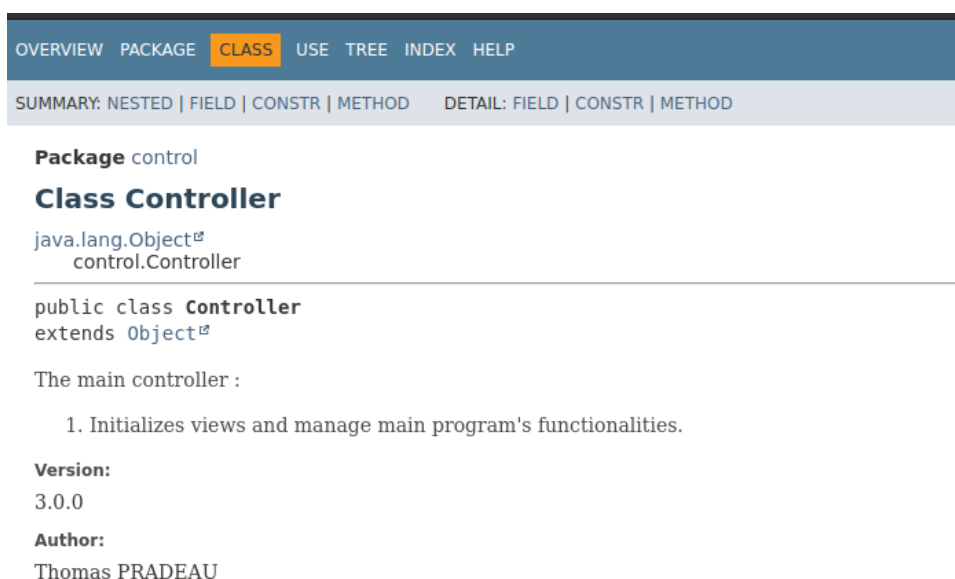
On retrouve la documentation dans un répertoire « Javadoc » à la racine du projet, dans lequel on trouve le fichier « index.html », tel que le montre la figure ci-dessous :



OVERVIEW PACKAGE CLASS USE TREE INDEX HELP	
Packages	
Package	Description
config	
control	
model	
utils	
view	
view.component	

Figure 54 Fichier index.html de la documentation.

On retrouve les différents packages du projet, dans lesquels il est possible de naviguer pour retrouver les différentes classes. La figure ci-dessous montre la documentation de la classe Controller.java :



OVERVIEW PACKAGE CLASS USE TREE INDEX HELP	
SUMMARY: NESTED FIELD CONSTR METHOD DETAIL: FIELD CONSTR METHOD	
Package control Class Controller java.lang.Object [Ⓓ] control.Controller	
<pre>public class Controller extends Object[Ⓓ]</pre>	
The main controller : <ol style="list-style-type: none"> 1. Initializes views and manage main program's functionalities. 	
Version: 3.0.0	
Author: Thomas PRADEAU	

55. Figure : Début de la documentation de la classe Controller.java

Tous les commentaires du code sont regroupés dans ces fichiers et mis en forme. Il est également possible d'utiliser des balises HTML directement dans les commentaires Java.

Déploiement

Concernant la phase de déploiement, nous verrons deux méthodes de déploiement d'une archive Jar, selon deux cas de figure précis.

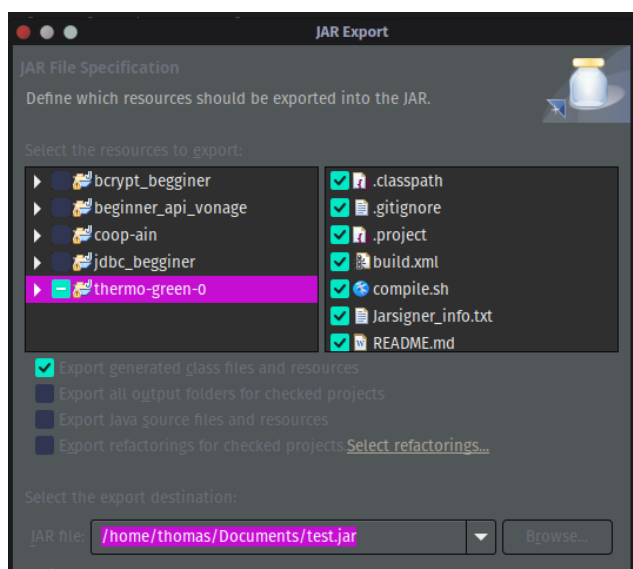
Dans un premier temps, nous verrons le déploiement d'une archive Jar non exécutable contenant l'intégralité des sources non compilées du projet. Enfin nous verrons le déploiement via une archive Jar exécutable, cette fois-ci ne contenant que les sources et bibliothèques nécessaires et compilées.

Archive jar non exécutable

Cette archive contient l'ensemble des sources non compilées du projet, et est à destination de l'équipe de développement.

Il s'agit d'une archive qu'il suffit de « dézipper » pour en extraire toutes ses sources. Pour créer une telle archive sous Eclipse, l'on se rend sous « Exporter », puis « Jar file ». Cette option permet d'exporter le projet, sélectionné sous forme d'archive Jar non exécutable.

La figure suivante montre les paramètres d'exportation, dont les fichiers à exporter :



56. Figure : Fichiers à exporter

Remarque : Il est possible de sélectionner des fichiers d'autres projets dans le même espace de travail.

On donne ensuite l'emplacement et le nom de l'archive, ici « export.jar ».

Ce type d'archive permet de faciliter la diffusion d'un projet au sein d'une équipe de développement, puisqu'il suffit ensuite d'importer le projet directement depuis l'archive.

Remarque : On préférera tout de même utiliser un outil de gestion de version et de travail collaboratif tel que Git pour ce cas d'utilisation. Ce type d'outils permettant de synchroniser et identifier tous les changements apportés au projet.

Archive Jar exécutable

Ensuite, l'archive Jar exécutable permet, comme son nom l'indique, d'être exécutée pour permettre de lancer directement le programme, comme un .exe par exemple.

Ce type d'archive sera cependant ciblée pour les utilisateurs finaux, puisque les sources sont compilées et que l'archive ne contient que le strict nécessaire pour faire fonctionner le programme. Alors que l'archive non exécutable destinée à l'équipe de développement contient l'intégralité du projet, y compris la documentation, fichier de mise en place, etc.

Une archive Jar exécutable se différencie principalement par la présence d'un fichier « Manifest », présent dans un répertoire dédié à la racine d'une archive Jar exécutable. Le manifeste est généralement stocké dans un fichier appelé "MANIFEST.MF" à l'intérieur du répertoire spécial "META-INF" à l'intérieur de l'archive JAR.

Le manifeste peut contenir différentes informations, mais l'une des utilisations les plus courantes est de spécifier la classe principale qui doit être exécutée lorsque l'archive JAR est lancée comme une application Java autonome.

Voici un exemple simple de fichier Manifest :

```
Manifest-Version: 1.0  
Main-Class: control.Start
```

Cet exemple stipule que la classe principale se nomme "Start" et se trouve dans un package nommé "control".

Sous Eclipse, il est possible d'exporter un projet sous forme d'archive Jar exécutable en se rendant sous « Exporter », puis « JAR exécutable ».

Attention : Ici, il n'est **pas possible** de sélectionner les fichiers à mettre dans l'archive, voici les fichiers mis automatiquement dans l'archive par Eclipse :

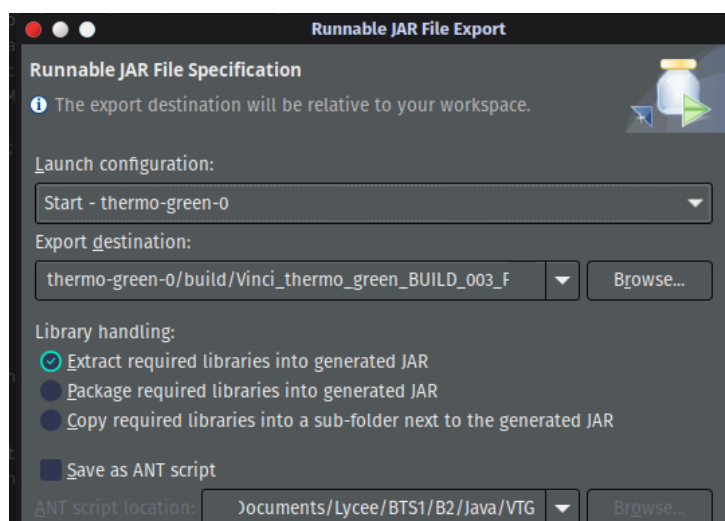
- Sources du projet compilées (src/*. java)
- Bibliothèques référencées (Build path)
- Contenu des dossiers ressources

Remarque : Les dossiers ressources sont des dossiers à la racine d'un projet qu'il est possible de marquer comme « dossier ressource » sous Eclipse en faisant un clic droit dessus, puis « Build path » et enfin « Utiliser en tant que dossier Source ».

Cela aura pour effet de dire à Eclipse de prendre le **contenu** de ces dossiers et de le mettre dans l'archive Jar exécutable générée par la suite.

Cette fonctionnalité est très utile lorsque l'on souhaite ajouter à l'archive des ressources comme des images ou des fichiers de configuration.

On finalise ensuite l'exportation en précisant la classe de démarrage, puis l'emplacement et le nom de l'archive générée. Tel que le montre la figure ci-dessous :



57. Figure : Paramètres d'exportation d'une archive Jar exécutable.

Remarque : On sélectionne la première option qui permet d'ajouter les bibliothèques nécessaires à l'intérieur de l'archive. Il est également possible de les mettre à l'extérieur ou dans un package dédié dans l'archive.

Signature et certification

L'étape suivante dans le déploiement de l'archive aux utilisateurs et la signature et la certification de celle-ci.

La **signature** permet de vérifier que le contenu ainsi que l'archive elle-même n'ont pas été modifiés après la signature. Cela permet d'éviter qu'un tiers remplace les sources du programme afin d'y ajouter une backdoor par exemple.

La **certification** permet d'attester l'identité du signataire du fichier. Un certificat est émis par une autorité de certification et permet d'assurer que le fichier provienne bien de cette personne ou organisation.

Le JDK offre divers outils permettant de générer un certificat et d'imposer une signature sur une archive Jar.

Préparations

Afin de générer un certificat, il est nécessaire dans un premier temps de générer une paire de clés privée et publique. Cette paire nous servira pour générer un certificat.

Un outil nommé « keytool » disponible dans le JDK permet de gérer tout ce qui est en lien avec la signature et la certification, pour générer un couple de clés, on utilise la commande suivante :

```
> keytool -genkeypair -keyalg RSA -keysize 2048 -alias cert_keys -  
keystore ca. keystore -validity 3650
```

Cette commande permet de générer des clés de 2048 bits de long avec l'algorithme RSA, qui portent comme nom « cert_keys » dans un keystore nommé « ca. keystore ».

L'utilitaire générera également en plus des clés privée et publique, le certificat associé. Lors de la génération, il est nécessaire de spécifier certaines informations personnelles pour créer le certificat. Le certificat créé sera donc un certificat dit « autosigné ».

Remarque : Si le keystore n'existe pas, celui-ci sera créé et un mot de passe sera demandé pour permettre le chiffrement de celui-ci.

Un keystore est un fichier qui stocke des couples de clés de manière sécurisés par le biais d'un mot de passe.

Exportation du certificat

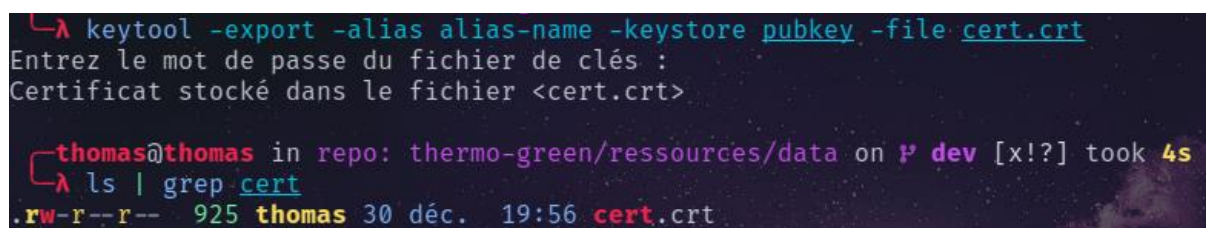
Pour pouvoir utiliser le certificat pour signer et certifier l'archive, il est nécessaire d'exporter celui-ci du keystore. Pour cela, on utilise la commande suivante :

```
> keytool -export -alias cert_keys -keystore ../ressources/data/ca.keystore -file cert.crt
```

Cette commande va extraire le certificat généré précédemment depuis le keystore et créer un fichier « cert.crt » qui contiendra le certificat.

Remarque : Pour exporter le certificat, on précisera l'alias, soit le nom donné à la paire de clés contenue dans le keystore.

La figure suivante montre l'exécution de la commande :



```

λ keytool -export -alias alias-name -keystore pubkey -file cert.crt
Entrez le mot de passe du fichier de clés :
Certificat stocké dans le fichier <cert.crt>

thomas@thomas in repo: thermo-green/ressources/data on p dev [x!?] took 4s
λ ls | grep cert
.rw-r--r-- 925 thomas 30 déc. 19:56 cert.crt
  
```

58. Figure : Exportation du certificat depuis le keystore

La commande « ls » ci-dessus montre le fichier « cert.crt » généré suite à la commande précédente. Celui-ci étant le certificat qui sera utilisé pour certifier l'archive à publier.

Vérification du certificat

Pour pouvoir utiliser le certificat sur une archive Jar, il est nécessaire de l'ajouter à un keystore un peu spécial. En effet, il est possible d'utiliser le certificat généré précédemment tel quel pour certifier l'archive.

Pour cela, on utilise la commande suivante :

```
> jarsigner -keystore pubkey -storepass password archive_file alias-name
```

Remarque : On précise uniquement le keystore à utiliser ainsi que le mot de passe, AINS que le nom de la paire de clés a utilisé.

Cependant, l'orque nous souhaiteront vérifier la certification de l'archive, l'erreur illustrée ci-dessous apparaîtra :

Jar verified.

Warning:

This jar contains entries whose certificate chain is invalid. Reason: PKIX path building failed: sun.security.provider.certpath.SunCertPathBuilderException: unable to find valid certification path to requested target
This jar contains entries whose signer certificate is self-signed.
This jar contains entries whose signer certificate will expire within six months.
This jar contains signatures that do not include a timestamp. Without a timestamp, users may not be able to validate this jar after any of the signer certificates expire (as early as 2024-02-25).

Re-run with the `-verbose` and `-certs` options for more details.

Cette erreur indique que le l'archive est bien signée et certifiée, mais que le certificat utiliser n'a pas pu être validé et est autosigné. En d'autres termes, cela signifie que le certificat utilisé n'a aucune réelle valeur.

Pour outrepasser cela, il est nécessaire d'importer notre certificat dans un keystore dédié aux certificats valide et vérifiable provenant d'organismes de certification. Cela permet de dire au JDK que le certificat utilisé a été émis d'une source sûre.

Ce keystore se nomme « cacert » (Certification Authorities Certificates), soit les certificats provenant d'autorité de certification. On peut y ajouter notre certificat avec cette commande :

`sudo keytool -import -alias alias-name -keystore /usr/lib/jvm/java-17-openjdk/lib/security/cacerts -file cert.crt`

Remarque : Cette commande doit être adaptée selon le système d'exploitation utilisé, dans ce cas il s'agit de Linux.

Ici, on importe le fichier « cert.crt », dans le keystore « cacerts ».
Pour valider l'opération, il est nécessaire de saisir le mot de passe de ce keystore. Par défaut, il s'agit de « changeit » (**changez-le**), il est clair qu'il faut changer ce mot de passe.

On peut désormais procéder à la signature et à la certification de notre archive.

Finalisation de la certification

On peut désormais finaliser la certification avec la commande suivante :

```
> jarsigner -keystore pubkey -storepass password archive_file alias-name
```

Remarque : On précise uniquement le keystore à utiliser ainsi que le mot de passe, AINS que le nom de la paire de clés a utilisé.

On pourra ensuite vérifier la certification avec la commande suivante :

```
> jarsigner -verify archive_file
```

Normalement, l'or de la vérification, l'outil jarsigner va vérifier la présence du certificat dans le keystore « cacerts » qui contient tous les certificats provenant d'autorité de certification.

Cela va permettre d'établir une chaîne de vérification et donc de pouvoir vérifier la validité du certificat. La figure ci-dessous présente la sortie de la commande précédente :

jar verified.

Warning:

This jar contains entries whose signer certificate is self-signed.

Re-run with the -verbose and -certs options for more details.

Le seul avertissement restant nous dit que l'archive contient des entrées signées avec un certificat autosigné, ce qui est parfaitement normal dans notre cas puisque nous avons créé nous-mêmes notre certificat.

Sinon, l'erreur concernant la chaîne de certification PKIX invalide à bien disparu. Autrement dit, la certification a été correctement vérifiée et validée.

Vérification des signatures dans l'archive

On peut vérifier qu'une archive Jar est bien signée et certifiée en l'ouvrant et en vérifiant la présence de certains fichiers dans le répertoire « META-INF », situé à la racine de l'archive.

Si l'archive est bien signée et certifiée, l'on devrait trouver deux fichiers, tels que « ALIAS - NA.RSA » et « ALIAS-NA.SF ».

Le fichier **ALIAS-NA.RSA** contient la clé privée et le certificat utilisé pour signer et certifier l'archive.

Le fichier **ALIAS-NA.SF** quant à lui contient toutes les empreintes générées pour chaque fichier constituant l'archive.

Et c'est ici que prend tout l'intérêt de ce procédé. Les empreintes des fichiers sont générées selon le contenu du fichier et le certificat utilisé. Autrement dit, si l'on essaie de modifier le contenu d'un des fichiers de l'archive, l'on ne pourrait pas régénérer la bonne empreinte si l'on ne pose de pas la clé privée utilisée l'or de la signature.

De cette manière, on utilise donc la clé publique pour vérifier l'identité du signataire et le statut de chaque fichier de l'archive.

Si le **certificat est invalide** ou alors **un ou plusieurs fichiers de l'archive ont été modifiés (s)**, alors l'outil « jarsigner » avec l'option « - verify » nous retrouverait une erreur.

Bibliographie

Vous pourrez retrouver la bibliographie consultée pour l'élaboration de ce Handbook dans le répertoire « biblio » du build, ou directement en ligne, via les URLs suivantes :

Table des illustrations

Figure 1 : Modélisation des classes métier	5
Figure 2 : Diagramme d'expérience utilisateur	5
Figure 3 : Diagramme de séquence en boîte noire	6
Figure 4 : Diagramme de séquence en boîte noire du changement de mot de passe ...	6
Figure 5 : Aperçu du diagramme de séquence complet de l'application	7
Figure 6 : Architecture MVC	8
Figure 7 : Mise à jour d'une mesure à titre d'illustration	9
Figure 8 : Insertion d'une mesure à titre d'illustration	9
Figure 9 : Attributs de la classe modèle mesure	10
Figure 10 : constructeur de la classe modèle mesure	10
Figure 11 : Définition de la méthode « getCelsius() » de la classe modèle mesure	10
Figure 12 : Attributs de la classe stade	11
Figure 13 : Second constructeur de la classe stade	11
Figure 14 : Attributs de la classe utilisateur	12
Figure 15 : Méthode updatePassword de la classe utilisateur	12
Figure 16 : Implémentation de la vue console sous Java Swing	14
Figure 17 : Partie permettant de changer de stade en cours de visualisation.	14
Figure 18 : Vue de connexion	15
Figure 19 : Message d'erreur si les identifiants sont vides.	15
Figure 20 : Message d'erreur si les identifiants sont invalides.	15
Figure 21 : Visualisation de la saisie du mot de passe.....	15
Figure 22 : Vue changement du mot de passe.	16
Figure 23 : Visualisation d'un mot de passe valide.....	16
Figure 24 : Erreur de vérification de sécurité des mots de passe.	16
Figure 25 : Instanciation du contrôleur au démarrage de l'application	18
Figure 26 : Instanciation des vues et du modèle utilisé.	18
Figure 27 : Méthode submitLogin du contrôleur.	19
Figure 28 : Méthode « filtrerLesMesures » permettant de filtrer les mesures selon les critères de l'utilisateur.....	20
Figure 29 : Vue console utilisant la fonctionnalité de filtrage.	20
Figure 30 : Contrôle des valeurs de débord.	21
Figure 31 : Températures hors des valeurs définies.....	21
Figure 32 : Script de création des tables avec leurs relations.	22
Figure 33 : Aperçu de la structure de la base de données.	24
Figure 34 : Attributs de la classe DatabaseHelper	25
Figure 35 : Fichier contenant les informations de connexion à la base de données.	26
Figure 36 : Lecture du fichier et création de la chaîne de connexion à la base de données.....	27
Figure 37 : Séparation de la chaîne et ouverture de la connexion.	27
Figure 38 : Connexion d'un utilisateur	28
Figure 39 : Erreur, refus de connexion	28
Figure 40 : Exemple de hash avec l'algorithme BCrypt.....	29
Figure 41 : Vue de changement du mot de passe.	30
Figure 42 : Validation du nouveau mot de passe avec Passay.	30
Figure 43 : Définition des règles de validation du mot de passe.	31
Figure 44 : Partie du fichier de traduction française.....	32
Figure 45 : Énumération EError.....	34
Figure 46 : Traitement du cas d'erreur numéro 1.....	35
Figure 47 : Journalisation d'un crash de l'application.....	35
Figure 48 : Crédits gratuits à la création d'un compte.....	36
Figure 49 : Paires de clés nécessaires pour communiquer avec l'API.	36
Figure 50 : Instanciation de la classe VonageClient	37
Figure 51 : Implémentation de l'envoi d'un SMS par le client Vonage.	37
Figure 52 : Récapitulatif des messages envoyés par l'API.	38
Figure 53 : Forme de la documentation sous Java	39
Figure 54 Fichier index.html de la documentation.	40

55. Figure : Début de la documentation de la classe Controller.java	40
56. Figure : Fichiers à exporter.....	41
57. Figure : Paramètres d'exportation d'une archive Jar exécutable.	43
58. Figure : Exportation du certificat depuis le keystore	45