Lors de cette nouvelle étude, je vous propose de voir comment gérer des bases de données avec la prise en compte de requêtes **SQL**. Cette étude ne sera pas exhaustive, mais nous permettra de bien voir comment réaliser des applications, sachant que nous utiliserons des bases de données de type **SQLITE**.

*Je rappelle que **SQLITE** n'utilise pas de serveur à proprement dit, mais permet de stocker l'ensemble des données dans un seul fichier avec toutefois une structure interne qui permet de retrouver toutes les informations rapidement avec des requêtes **SQL** classiques.*

Ce système est très souvent le plus adapté pour de toutes petites bases de données, puisque nous n'avons pas à gérer la communication réseau. Par contre, avec cette technique, il s'agit essentiellement d'un stockage local au système numérique qui possède l'application de gestion.

Pour commencer, je vous invite à savoir comment gérer les dates et les heures, puisque ces types d'informations sont souvent présents dans les bases de données.

Gestion du temps (date et heure)

Pour gérer le temps, il existe plusieurs solutions, notamment celle que nous connaissons déjà avec la caisse « **time** ». Toutefois, afin de pouvoir maîtriser le formatage des dates en prenant en compte le pays, la caisse « **chrono** » me paraît plus adaptée pour les bases de données.

Cargo.toml

```
[package]
name = "test-rust"
version = "0.1.0"
authors = ["manu"]
edition = "2018"

[dependencies.chrono]
version = "0.4.19"
features = ["unstable-locales"]
```

Récupération de la date et de l'heure actuelle

```
use chrono::prelude::*;

fn main() {
  let aujourdhui = Local::now();
  let date = aujourdhui.format_localized("%A, %d %B %Y", Locale::fr_FR);
  let heure = aujourdhui.format_localized("%H:%M:%S", Locale::fr_FR);
  let date_standard = aujourdhui.format_localized("%d-%m-%Y", Locale::fr_FR);

  println!("{}", aujourdhui);
  println!("{}", aujourdhui.date());
  println!("{}", aujourdhui.time());
  println!("{}", date);
  println!("{}", heure);
  println!("{}", date_standard);
}
```

Résultat

```
2021-09-07 17:37:28.482471033 +02:00
2021-09-07+02:00
17:37:28.482471033
mardi, 07 septembre 2021
17:37:28
07-09-2021
```

*Pour prendre en compte toutes les informations associées à l'heure et à la date, il existe l'entité qui porte bien son nom, **DateTime**, structure générique qui possède les méthodes respectives **date()** et **time()** ainsi que **format_localized()** qui permet de formater et donc de personnaliser votre information sur le temps, en utilisant les bons formateurs associés.*

*Pour connaître la date et l'heure actuelle en prenant en compte le pays concerné, il existe la structure **Local** qui possède la méthode statique **now()**.*

*Dans le même ordre d'idée, les constantes relatives au pays sont proposée dans l'entité **Locale**. Les variables de type **DateTime<Local>** comme **aujourdhui** peuvent utiliser la méthode **format_localized()** pour visualiser ou enregistrer des dates et des heures personnalisées.*

Mesurer des durées et calculer des temps écoulés

Toujours à l'aide de cette librairie « **chrono** », nous pouvons profiter de l'occasion pour connaître le nombre de jours existants entre deux dates ou alors d'évaluer un temps d'exécution entre deux mesures de temps.

*Nous passons alors par la structure **Duration** qui est automatiquement prise en compte lorsque nous soustrayons deux unités de temps, soit entre deux dates, soit entre deux heures, soit les deux.*

*Cette structure implémente un certain nombre de méthodes pour retrouver le nombre de jours écoulés **num_days()**, le nombre de nanosecondes écoulées **num_nanoseconds()** ou le nombre de microsecondes **num_microseconds()**, etc.*

Création d'une date précise et évaluation de durées

```
use chrono::prelude::*;

fn main() {
  let maintenant = Local::now();
  println!("{}", maintenant.format_localized("%A, %d %B %Y", Locale::fr_FR));

  let naissance = Local.ymd(1959, 10, 1);
  println!("{}", naissance.format_localized("%A, %d %B %Y", Locale::fr_FR));

  let temps_ecoule = maintenant.date() - naissance;
  let age = (temps_ecoule.num_days() as f64 / 365.25) as u32;
  println!("Âge : {} ans", age);

  let temps_execution = Local::now() - maintenant;
  println!("Temps d'exécution : {} ns", temps_execution.num_nanoseconds().unwrap());
  println!("Temps d'exécution : {} µs", temps_execution.num_microseconds().unwrap());
}
```

Résultat

```
vendredi, 10 septembre 2021
jeudi, 01 octobre 1959
Âge : 61 ans
Temps d'exécution : 264922 ns
Temps d'exécution : 264 µs
```

Élaboration d'une structure en collaboration avec la base de données

Il est souvent judicieux de proposer des structures, avec l'implémentation des méthodes associées, qui vont représenter chacune des tables de la base de données, l'une est une image de l'autre. Ultérieurement, nous mettrons en œuvre une base de données qui permettra de stocker et de recenser les différents rendez-vous enregistrés.

*Cette base de données possèdera une seule table et avant de nous intéressez à la base elle-même, je vous propose de réaliser toute l'architecture associée à la structure représentative de la table correspondante. Nous élaborerons l'implémentation des requêtes **SQL** dans le prochain chapitre.*

*Pour l'instant, seule la structure nommée **RendezVous** nous intéresse. Par anticipation, elle devra posséder **l'identification** de l'enregistrement, **l'intitulé** (titre) le **jour** et **l'heure** du rendez-vous.*

Pour que l'implémentation globale du projet soit agréable, je vous propose de le découper en plusieurs fichiers, avec pour l'instant l'élaboration de la structure avec ses méthodes et ses fonctions associées dans un fichier spécifique. Dans la suite de l'étude, nous ajouterons un autre fichier pour la gestion globale de la base de données.

*Pour que cela fonctionne correctement, nous devons mettre en œuvre la notion de **modules**, correspondant aux différents fichiers sources concernés. Pour qu'un fichier source puisse utiliser un autre fichier source, vous devez le préciser d'une par une déclaration du module concerné avec le mot réservé **mod**, et ensuite désigner les éléments de ce module que vous souhaitez utiliser grâce au mot réservé **use**.*

Cargo.toml

```
[package]
name = "test-rust"
version = "0.1.0"
authors = ["manu"]
edition = "2018"

[dependencies.chrono]
version = "0.4.19"
features = ["unstable-locales"]
```

rendez_vous.rs

```
use chrono::prelude::*;
use std::io::{stdout, Write, stdin};

// Zone publique
#[derive(Debug)]
pub struct RendezVous {
  pub id: i32,
  pub titre: String,
  pub date: DateTime<Local>
}

impl RendezVous {
  pub fn now() -> Self {
```

```
    RendezVous {
      id: 0,
      titre: String::new(),
      date: Local::now()
    }
  }
  pub fn affiche(&self) {
    let quand = self.date.format_localized("le %A %d %B %Y à %Hh%Mmn", Locale::fr_FR);
    println!("{} {}", self.titre, quand);
  }
  pub fn heure(&self) -> String {
    self.date.format_localized("%Hh%Mmn", Locale::fr_FR).to_string()
  }
  pub fn jour(&self) -> String {
    self.date.format_localized("%A %d %B %Y", Locale::fr_FR).to_string()
  }
}

pub fn saisie() -> RendezVous {
  let mut rdv = RendezVous::now();
  rdv.titre = saisie_titre();
  rdv.date = saisie_date();
  rdv
}

// fonctions privées
fn saisie_titre() -> String {
  let mut saisie = String::new();
  print!("Intitulé de votre rendez-vous : ");
  stdout().flush().unwrap();
  stdin().read_line(&mut saisie).unwrap();
  saisie.trim().to_string()
}

fn saisie_date() -> DateTime<Local> {
  let aujourdhui = Local::now();
  loop {
    let mut saisie = String::new();
    print!("Saisissez votre date et votre heure (jj-mm-aaaa hh:mn) : ");
    stdout().flush().unwrap();
    stdin().read_line(&mut saisie).unwrap();

    match Local.datetime_from_str(saisie.trim_end(), "%d-%m-%Y %H:%M") {
      Ok(jour) => if jour<=aujourdhui { println!("Date dépassée !") }
                  else { return jour },
      Err(_) => println!("Saisie incorrecte, veuillez recommencer")
    }
  }
}
```

main.rs

```
mod rendez_vous;

use rendez_vous::{RendezVous, saisie};
use chrono::prelude::*;

fn main() {
  let rdv = RendezVous {
    id: 0,
    titre: "Dentiste".to_string(),
    date: Local.ymd(2021, 10, 20).and_hms(15, 30, 0)
  };
  println!("{:?}", rdv);
  rdv.affiche();
  println!("Heure : {}", rdv.heure());
  println!("Jour : {}", rdv.jour());

  let nouveau = saisie();
  nouveau.affiche();
}
```

Résultat

```
RendezVous { id: 0, titre: "Dentiste", date: 2021-10-20T15:30:00+02:00 }
Dentiste le mercredi 20 octobre 2021 à 15h30mn
Heure : 15h30mn
Jour : mercredi 20 octobre 2021
```

```
Intitulé de votre rendez-vous : Ophtalmologue
Saisissez votre date et votre heure (jj-mm-aaaa hh:mn) : 22-11-2021
Saisie incorrecte, veuillez recommencer
Saisissez votre date et votre heure (jj-mm-aaaa hh:mn) : 22-07-2021 17:45
Date dépassée !
Saisissez votre date et votre heure (jj-mm-aaaa hh:mn) : 22-11-2021 17:45
Ophtalmologue le lundi 22 novembre 2021 à 17h45mn
```

Création d'une base de données et création de notre première table

Nous pouvons maintenant rentrer dans le vif du sujet et voir comment gérer une base de données, avec dans un premier temps juste la création de la base avec la table associée. Pour cela, nous utilisons une librairie adaptée « **rusqlite** ». Il existe d'autres librairies, mais celle-ci est particulièrement adaptée pour passer les informations d'une structure vers la table correspondante et inversement (**wrapping**).

*L'implémentation d'une base de données se fait au travers de la structure **Connection** qui possède un certain nombre de méthodes pour créer ou ouvrir une base de données, pour proposer des requêtes **SQL** et enfin pour permettre rapidement de pouvoir récupérer les informations souhaitées.*

- ***open_in_memory() : méthode statique :** il est possible de gérer une base de données directement en mémoire pour l'exploiter rapidement et l'enregistrer après coup. Ceci n'est valable que pour la première construction ou lorsque vous ne voulez pas générer de fichiers. Dans ce cas là, il s'agit juste de stocker des informations temporaires qui sont très facile à récupérer après coup.*
- ***open(nom du fichier) : méthode statique :** c'est certainement la méthode que vous utiliserez la plus souvent pour ouvrir une base de données intégrée dans un fichier. Si le fichier n'existe pas, la base de données est alors créée avec un fichier qui porte exactement le même nom que l'argument.*
- ***execute(requête, paramètres) :** la méthode qui sera certainement la plus utilisée et qui permet de réaliser toutes les requêtes **SQL** en précisant les paramètres éventuels pour générer des textes paramétrées. Si la requête est simple et non paramétrées, vous placez alors un tableau vide « **[ ]** ».*
- ***execute_batch(requêtes) :** moins utile, elle permet de réaliser plusieurs requêtes consécutives en une seule fois sans pouvoir proposer de paramètres.*
- ***prepare(requête) :** propose également l'élaboration d'une requête **SQL**, comme avec **execute()**, mais celle-ci est dite préparée, c'est-à-dire paramétrée. C'est celle que nous utilisons pour récupérer des valeurs stockées dans la base de données. Avec les requêtes de type « **SELECT … FROM ….** ».*

*Pour créer notre première base de données avec la génération de la table **RDV**, je vous propose d'organiser notre projet avec une architecture qui permet de séparer les différents constituants.*

*À ce sujet, je vous propose de créer un **trait** nommée **BDD** qui nous donne l'ossature habituelle d'une gestion de base de données quelle quel soit, avec des méthodes adaptées aux différentes requêtes souhaitées.*

*Mise à part l'ouverture de la base de données dont l'écriture est souvent générique, nous implémenterons ensuite les différentes méthodes du **trait** par l'intermédiaire de la structure **RendezVous** afin de proposer des requêtes adaptées.*

Toute cette démarche se fait au travers de fichiers sources séparés afin que cela soit plus facile à implémenter. Cette organisation nous permet de bien maîtriser les différents phases de fonctionnement. Voici ci-dessous le fichier de configuration avec l'ensemble des fichiers sources. Par contre, le source « rendez_vous.rs » n'étant pas modifier, il ne sera pas présent ci-dessous bien qu'il existe bien évidemment.

Cargo.toml

```
[package]
name = "test-rust"
version = "0.1.0"
authors = ["manu"]
edition = "2018"

[dependencies.chrono]
version = "0.4.19"
features = ["unstable-locales"]

[dependencies.rusqlite]
version = "0.25.3"
features = ["bundled"]
```

*Afin de prendre en compte cette librairie « **rusqlite** », nous devons proposer la dépendance associée, sachant qu'elle peut-être adaptée quelque soit le système d'exploitation. Nous pouvons ainsi proposer ce projet pour une **Raspberry** par exemple avec exactement les mêmes codes sources.*

bdd.rs

```
use rusqlite::Connection;
use std::fs::File;

pub trait BDD {
  fn ouverture(&self, nom: &str) -> Connection {
    let ouverture = Connection::open(nom);
```

```
  match ouverture {
   Ok(bdd) => {
    println!("Ouverture de la base '{}'", nom);
    let fichier = File::open(nom).unwrap();
    if fichier.metadata().unwrap().len() == 0 { self.creation_table(&bdd) }
    bdd
   },
   Err(_) => panic!("Impossible d'ouvrir la base de données ({fichier})", fichier= nom)
  }
 }

 fn creation_table(&self, bdd: &Connection);
 fn enregistrer(&self, bdd: &Connection);
 fn recuperer(&mut self, bdd: &Connection, id: u32);
 fn modifier(&self, bdd: &Connection);
 fn supprimer(&self, bdd: &Connection);
}
```

*Nous aurions pu proposer de simples fonctions pour résoudre les **requêtes SQL** associées à la structure **RendezVous**, mais il me semble plus judicieux de mettre en œuvre un **trait** spécifique sur la gestion de base de données que nous pourrons utiliser pour un tout autre projet avec une ou plusieurs structures totalement différentes.*

*Comme nous l'avons déjà précisé, l'ouverture d'une base de données est générique, c'est-à-dire que le code source est similaire quelque soit la base de données utilisée. Du coup, nous pouvons proposer la fonction **ouverture()** par défaut directement dans la déclaration du **trait BDD**.*

*Lors de l'ouverture de la base de données, si le fichier n'existe pas, il est alors créé automatiquement. De ce fait, la fonction **creation_table()** est également activée. Si le fichier existait déjà, bien sûr la création de la table n'est pas pris en compte.*

rdv_bdd.rs

```
use crate::bdd::BDD;
use crate::rendez_vous::RendezVous;
use rusqlite::{params, Connection, Result};

impl BDD for RendezVous {
 fn creation_table(&self, bdd: &Connection) {
  match bdd.execute("CREATE TABLE IF NOT EXISTS RDV (
                     id INTEGER PRIMARY KEY,
                     titre VARCHAR(40),
                     date TIMESTAMP)", []) {
   Ok(_)  => println!("Création de la table"),
   Err(e) => println!("Difficultés pour créer la table ({})", e)
  }
 }
 fn enregistrer(&self, bdd: &Connection)                { todo!() }
 fn recuperer(&mut self, bdd: &Connection, id: u32) {  todo!()  }
 fn modifier(&self, bdd: &Connection)                  { todo!()  }
 fn supprimer(&self, bdd: &Connection)                 {  todo!()  }
}
```

*Pour ce chapitre, l'objectif est de créer la table **RDV** qui est une image de la structure **RendezVous**. Vous remarquez que ce fichier source « **rdv_bdd.rs** » recense toutes les méthodes qui respecte le trait **BDD** et qui sont rattachées à la structure **RendezVous**.*

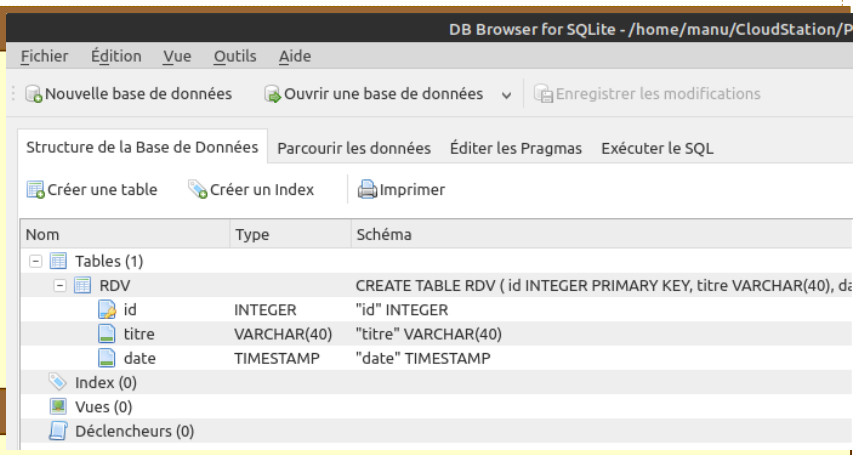
*Pour créer cette table **RDV**, nous proposons une simple requête **SQL** classique avec le texte nécessaire sans paramétrage particulier à l'aide de la méthode **execute()**. Du coup, pour le deuxième argument, vous proposer un tableau statique vide **[ ]**, ou alors, vous pouvez proposer la constante **NO_PARAMS**.*

main.rs

```
mod rendez_vous;
mod bdd;
mod rdv_bdd;

use chrono::prelude::*;
use rendez_vous::{RendezVous, saisie};
use bdd::*;

fn main() {
 let rdv = RendezVous::now();
 rdv.affiche();
 let connexion = rdv.ouverture("rdv.db3");
}
```



Résultat

```
le samedi 18 septembre 2021 à 14h59mn
Ouverture de la base 'rdv.db3'
Création de la table
```

Enregistrement d'un rendez-vous

Je vous propose de compléter nos actions de gestion de base de données petit à petit en prenant en compte chaque méthode respectant le **trait BDD**. Nous commençons dans ce chapitre par la méthode **enregistrer()**. Pour cela, nous pouvons de nouveau utiliser la méthode **execute()** de **Connection** comme tout-à-l'heure. Par contre, nous devons spécifier les paramètres pour bien prendre en compte le rendez-vous souhaité.

rdv_bdd.rs

```
use crate::bdd::BDD;
use crate::rendez_vous::RendezVous;
use rusqlite::{params, Connection, Result};

impl BDD for RendezVous {
 fn creation_table(&self, bdd: &Connection) {
  match bdd.execute("CREATE TABLE IF NOT EXISTS RDV (
                     id INTEGER PRIMARY KEY,
                     titre VARCHAR(40),
                     date TIMESTAMP)", []) {
   Ok(_)  => println!("Création de la table"),
   Err(e) => println!("Difficultés pour créer la table ({})", e)
  }
 }

 fn enregistrer(&self, bdd: &Connection) {
  match bdd.execute("INSERT INTO RDV (titre, date) VALUES(?1, ?2)",
                    params![self.titre, self.date.to_string()]) {
   Ok(_)  => println!("Rendez-vous enregistré"),
   Err(e) => println!("Requête incorrecte ({})", e)
  }
 }

 fn recuperer(&mut self, bdd: &Connection, id: u32) {  todo!()  }
 fn modifier(&self, bdd: &Connection)     {  todo!()  }
 fn supprimer(&self, bdd: &Connection)  {  todo!()  }
}
```

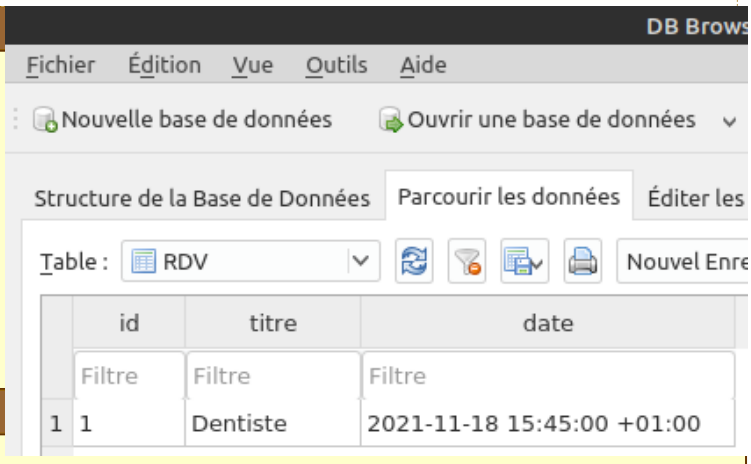
*Cette fois-ci, nous proposons une requête **SQL** paramétrée. Pour cela nous utilisons la macro **param![ ]** dont les éléments de la liste vont être associés au texte de la requête à l'endroit où se trouve des marqueurs spécifiques représentés par le caractère « **?** » suivi du numéro correspondant à l'ordre des arguments.*

main.rs

```
mod rendez_vous;
mod bdd;
mod rdv_bdd;

use chrono::prelude::*;
use rendez_vous::{RendezVous, saisie};
use bdd::*;

fn main() {
  let rdv = saisie();
  let bdd = rdv.ouverture("rdv.db3");
  rdv.enregistrer(&bdd);
}
```



Résultat

```
Intitulé de votre rendez-vous : Dentiste
Saisissez votre date et votre heure (jj-mm-aaaa hh:mn) : 18-11-2021 15:45
Ouverture de la base 'rdv.db3'
Création de la table
Rendez-vous enregistré
```

Récupérer et lire un rendez-vous enregistré

Une fois que les enregistrements sont effectués, nous pouvons les récupérer en donnant l'identifiant correspondant. Afin de résoudre cette action, nous implémentons la méthode **recuperer()** qui permet de compléter l'objet de type **RendezVous** en spécifiant l'identifiant passé en argument de la méthode.

rdv_bdd.rs

```
use crate::bdd::BDD;
use crate::rendez_vous::RendezVous;
use rusqlite::{params, Connection, Result};
use chrono::{DateTime, Local};

impl BDD for RendezVous {
```

```rust
 fn creation_table(&self, bdd: &Connection) {
  match bdd.execute("CREATE TABLE IF NOT EXISTS RDV (
                    id INTEGER PRIMARY KEY,
                    titre VARCHAR(40),
                    date TIMESTAMP)", []) {
    Ok(_)  => println!("Création de la table"),
    Err(e) => println!("Difficultés pour créer la table ({})", e)
  }
 }

 fn enregistrer(&self, bdd: &Connection) {
  match bdd.execute("INSERT INTO RDV (titre, date) VALUES(?1, ?2)",
              params![self.titre, self.date.to_string()]) {
    Ok(_)  => println!("Rendez-vous enregistré"),
    Err(e) => println!("Requête incorrecte ({})", e)
  }
 }

 fn recuperer(&mut self, bdd: &Connection, id: u32) {
  let mut requete = bdd.prepare("SELECT * FROM RDV WHERE id=?1").unwrap();
  let resultat = requete.query_row(params![id], |ligne| {
    let jour : String = ligne.get(2)?;
    Ok(RendezVous {
      id: ligne.get(0)?,
      titre: ligne.get(1)?,
      date: jour.parse::<DateTime<Local>>().unwrap()
    })
  });
  match resultat {
    Ok(rdv) => *self = rdv,
    Err(_) => println!("Cet enregistrement n'existe pas !")
  }
 }

 fn modifier(&self, bdd: &Connection)    { todo!() }
 fn supprimer(&self, bdd: &Connection) { todo!() }
}
```

*Cette méthode va modifier le contenu de l'objet **RendezVous** qui possède cette méthode. C'est pour cette raison que le premier paramètre est **&mut self**.*

*Pour récupérer un ou plusieurs enregistrements d'une table dans la base de données, nous passons par une nouvelle méthode **prepare()** de **Connection** à l'intérieur de laquelle vous spécifiez votre requête **SQL** qui peut être paramétrée (généralement des requête de type **SELECT**). Vu qu'il s'agit d'une requête préparée, elle n'est pas encore exécutée.*

*Cette méthode **prepare()** renvoie un objet de type **Result<Statement>**. **Statement** propose deux méthodes intéressantes pour résoudre la plupart des requêtes qui permettent de récupérer des informations. La première **query_row()** permet de récupérer un seul enregistrement. La deuxième **query_map()** récupère l'ensemble des occurrences de tous les enregistrement souhaités sous forme d'un itérateur.*

*Vu que nous avons besoin de récupérer un seul rendez-vous correspondant à l'identifiant choisi, la méthode qui nous intéresse ici est **query_row()**. Cette méthode a besoin de deux arguments, le premier pour spécifier les paramètres nécessaires à la requête paramétrée si tel est le cas, le deuxième pour implémenter une **clôture** qui résout le traitement à réaliser associé à la ligne de la table correspondant à l'enregistrement récupéré.*

*Le choix de l'identifiant peut ne pas correspondre à ce qui est réellement enregistré actuellement dans la base de données. Il paraît alors logique que la **clôture** de résolution du traitement renvoie une information de type **Result<_>**. Cela permet de savoir si notre recherche a aboutie ou pas.*

main.rs

```rust
mod rendez_vous;
mod bdd;
mod rdv_bdd;

use chrono::prelude::*;
use rendez_vous::{RendezVous, saisie};
use bdd::*;

fn main() {
  let mut rdv = RendezVous::now();
  let bdd = rdv.ouverture("rdv.db3");
  rdv.recuperer(&bdd, 1);
  rdv.affiche();
}
```

Résultat

```
Ouverture de la base 'rdv.db3'
```

Dentiste le jeudi 18 novembre 2021 à 15h45mn

Modifier un enregistrement

Il est bien sûr possible de modifier un enregistrement en passant par une requête **SQL** de type **UPDATE**. Nous devons souvent modifier un enregistrement lors de la récupération en voyant une erreur sur l'intitulé ou sur la date globale. Afin de garder l'identifiant de cet enregistrement, il est alors préférable de rajouter une méthode de saisie spécifique pour la structure **RendezVous** en parallèle de la fonction qui elle génère un nouveau rendez-vous complet.

rendez_vous.rs

```rust
use chrono::prelude::*;
use std::io::{stdout, Write, stdin};

// Zone publique
#[derive(Debug, Clone)]
pub struct RendezVous {
  pub id: i32,
  pub titre: String,
  pub date: DateTime<Local>
}

impl RendezVous {
  pub fn now() -> Self {
    RendezVous {
      id: 0,
      titre: String::new(),
      date: Local::now()
    }
  }
  pub fn affiche(&self) {
    let quand = self.date.format_localized("le %A %d %B %Y à %Hh%Mmn", Locale::fr_FR);
    println!("{} {}", self.titre, quand);
  }
  pub fn heure(&self) -> String {
    self.date.format_localized("%Hh%Mmn", Locale::fr_FR).to_string()
  }
  pub fn jour(&self) -> String {
    self.date.format_localized("%A %d %B %Y", Locale::fr_FR).to_string()
  }
  pub fn saisie(&mut self) {
    self.titre = saisie_titre();
    self.date = saisie_date();
  }
}

pub fn saisie() -> RendezVous {
  let mut rdv = RendezVous::now();
  rdv.titre = saisie_titre();
  rdv.date = saisie_date();
  rdv
}

// fonctions privées
fn saisie_titre() -> String {
  let mut saisie = String::new();
  print!("Intitulé de votre rendez-vous : ");
  stdout().flush().unwrap();
  stdin().read_line(&mut saisie).unwrap();
  saisie.trim().to_string()
}

fn saisie_date() -> DateTime<Local> {
  let aujourdhui = Local::now();
  loop {
    let mut saisie = String::new();
    print!("Saisissez votre date et votre heure (jj-mm-aaaa hh:mn) : ");
    stdout().flush().unwrap();
    stdin().read_line(&mut saisie).unwrap();

    match Local.datetime_from_str(saisie.trim_end(), "%d-%m-%Y %H:%M") {
      Ok(jour) => if jour<=aujourdhui { println!("Date dépassée !") }
                  else { return jour },
      Err(_) => println!("Saisie incorrecte, veuillez recommencer")
    }
  }
}
```

rdv_bdd.rs

```rust
use crate::bdd::BDD;
use crate::rendez_vous::RendezVous;
use rusqlite::{params, Connection, Result};
use chrono::{DateTime, Local};

impl BDD for RendezVous {
 fn creation_table(&self, bdd: &Connection) {
  match bdd.execute("CREATE TABLE IF NOT EXISTS RDV (
                     id INTEGER PRIMARY KEY,
                     titre VARCHAR(40),
                     date TIMESTAMP)", []) {
   Ok(_)  => println!("Création de la table"),
   Err(e) => println!("Difficultés pour créer la table ({})", e)
  }
 }

 fn enregistrer(&self, bdd: &Connection) {
  match bdd.execute("INSERT INTO RDV (titre, date) VALUES(?1, ?2)",
                    params![self.titre, self.date.to_string()]) {
   Ok(_)  => println!("Rendez-vous enregistré"),
   Err(e) => println!("Requête incorrecte ({})", e)
  }
 }

 fn recuperer(&mut self, bdd: &Connection, id: u32) {
  let mut requete = bdd.prepare("SELECT * FROM RDV WHERE id=?1").unwrap();
  let resultat = requete.query_row(params![id], |ligne| {
   let jour : String = ligne.get(2)?;
   Ok(RendezVous {
    id: ligne.get(0)?,
    titre: ligne.get(1)?,
    date: jour.parse::<DateTime<Local>>().unwrap()
   })
  });
  match resultat {
   Ok(rdv) => *self = rdv,
   Err(_) => println!("Cet enregistrement n'existe pas !")
  }
 }

 fn modifier(&self, bdd: &Connection) {
  match bdd.execute("UPDATE RDV SET titre=?1, date=?2 WHERE id=?3",
                    params![self.titre, self.date.to_string(), self.id]) {
   Ok(_)  => println!("Rendez-vous modifié"),
   Err(e) => println!("Requête incorrecte ({})", e)
  }
 }
 fn supprimer(&self, bdd: &Connection) {  todo!() }
}
```

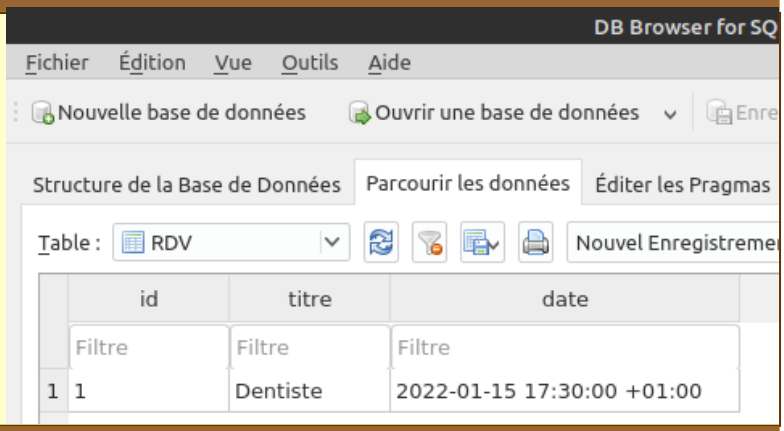
*Pour résoudre la méthode **modifier()**, nous n'avons pas besoin de connaître une méthode supplémentaire de **Connection**, il suffit de reprendre la méthode **execute()** avec les paramètres nécessaires à l'élaboration de la requête **SQL**.*

main.rs

```rust
mod rendez_vous;
mod bdd;
mod rdv_bdd;

use chrono::prelude::*;
use rendez_vous::{RendezVous, saisie};
use bdd::*;

fn main() {
 let mut rdv = RendezVous::now();
 let bdd = rdv.ouverture("rdv.db3");
 rdv.recuperer(&bdd, 1);
 rdv.affiche();
 rdv.saisie();
 rdv.modifier(&bdd);
}
```



Résultat

```
Ouverture de la base 'rdv.db3'
Dentiste le jeudi 18 novembre 2021 à 15h45mn
Intitulé de votre rendez-vous : Dentiste
```

```
Saisissez votre date et votre heure (jj-mm-aaaa hh:mn) : 15-01-2022 17:30
Rendez-vous modifié
```

*Jusqu'à présent, dans les requêtes paramétrées de la structure **Connection**, nous avons systématiquement proposé les formateurs «**?n** ». Il existe une alternative qui permet cette fois-ci de nommer les paramètres avec le préfixe « **:** », par exemple «**:id** ». Vous devez alors plutôt choisir la macro **named_params!()** à la place de **params!()**.*

***Attention !** Ce ne sont plus des crochets « **[ ]** » qui entourent les paramètres, mais des accolades « **{ }** ». La syntaxe utilisée pour chaque argument est une chaîne de caractères avec exactement le même nommage du paramètre suivi d'un deux-point « **:** » et de la valeur à soumettre. Cette fois-ci l'ordre n'a plus d'importance.*

rdv_bdd.rs

```
use rusqlite::{params, Connection, Result, named_params};
...

 fn modifier(&self, bdd: &Connection) {
  match bdd.execute("UPDATE RDV SET titre=:titre, date=:date WHERE id=:id",
                named_params! {":titre": self.titre,  ":date": self.date.to_string(),  ":id": self.id}) {
   Ok(_)  => println!("Rendez-vous modifié"),
   Err(e) => println!("Requête incorrecte ({})", e)
  }
 }
..
```

Supprimer un enregistrement

Nous pouvons supprimer un enregistrement spécifique. La requête **SQL** est généralement relativement simple avec très peu de paramètres, souvent l'identifiant suffit. Là aussi, la méthode **execute()** de **Connection** est adaptée pour ce type de traitement puisque nous n'avons pas besoin d'un résultat associé à cette requête.

*Pour la méthode **supprimer()** du trait **BDD**, nous avons le choix entre proposer un paramètre représentant l'identifiant ou pas. Les deux choix sont équivalent d'un point de vue logique. J'ai préféré ne pas mettre de paramètre avec la même idée que lors de la modification. C'est-à-dire que je préfère au préalable récupérer l'enregistrement pour vérifier le contenu avant la suppression.*

rdv_bdd.rs

```
use crate::bdd::BDD;
use crate::rendez_vous::RendezVous;
use rusqlite::{params, Connection, Result, named_params};
use chrono::{DateTime, Local};

impl BDD for RendezVous {
 fn creation_table(&self, bdd: &Connection) {
  match bdd.execute("CREATE TABLE IF NOT EXISTS RDV (
                id INTEGER PRIMARY KEY,
                titre VARCHAR(40),
                date TIMESTAMP)", []) {
   Ok(_)  => println!("Création de la table"),
   Err(e) => println!("Difficultés pour créer la table ({})", e)
  }
 }

 fn enregistrer(&self, bdd: &Connection) {
  match bdd.execute("INSERT INTO RDV (titre, date) VALUES(?1, ?2)",
                params![self.titre, self.date.to_string()]) {
   Ok(_)  => println!("Rendez-vous enregistré"),
   Err(e) => println!("Requête incorrecte ({})", e)
  }
 }

 fn recuperer(&mut self, bdd: &Connection, id: u32) {
  let mut requete = bdd.prepare("SELECT * FROM RDV WHERE id=?1").unwrap();
  let resultat = requete.query_row(params![id], |ligne| {
   let jour : String = ligne.get(2)?;
   Ok(RendezVous {
    id: ligne.get(0)?,
    titre: ligne.get(1)?,
    date: jour.parse::<DateTime<Local>>().unwrap()
   })
  });
  match resultat {
   Ok(rdv) => *self = rdv,
   Err(_) => println!("Cet enregistrement n'existe pas !")
  }
 }
```

```rust
fn modifier(&self, bdd: &Connection) {
  match bdd.execute("UPDATE RDV SET titre=:titre, date=:date WHERE id=:id",
                named_params! {":titre": self.titre,
                               ":date": self.date.to_string(),
                               ":id": self.id}) {
    Ok(_)  => println!("Rendez-vous modifié"),
    Err(e) => println!("Requête incorrecte ({})", e)
  }
 }

 fn supprimer(&self, bdd: &Connection) {
  match bdd.execute("DELETE FROM RDV WHERE id=:id", named_params! {":id": self.id}) {
    Ok(_)  => println!("Rendez-vous supprimé"),
    Err(e) => println!("Requête incorrecte ({})", e)
  }
 }
}
```

main.rs

```rust
mod rendez_vous;
mod bdd;
mod rdv_bdd;

use chrono::prelude::*;
use rendez_vous::{RendezVous, saisie};
use bdd::*;

fn main() {
  let mut rdv = RendezVous::now();
  let bdd = rdv.ouverture("rdv.db3");
  rdv.recuperer(&bdd, 2);
  rdv.affiche();
  rdv.supprimer(&bdd);
}
```

Table : RDV — Nouvel Enregistrement

	id	titre	date
	Filtre	Filtre	Filtre
1	1	Ophtalmologue	2021-12-18 15:00:00 +01:00
2	2	Dentiste	2022-05-15 14:15:00 +02:00
3	3	Réunion pédagogique	2022-01-23 08:00:00 +01:00

Table : RDV — Nouvel Enregistrement

	id	titre	date
	Filtre	Filtre	Filtre
1	1	Ophtalmologue	2021-12-18 15:00:00 +01:00
2	3	Réunion pédagogique	2022-01-23 08:00:00 +01:00

Résultat

```
Ouverture de la base 'rdv.db3'
Dentiste le dimanche 15 mai 2022 à 14h15mn
Rendez-vous supprimé
```

Liste des rendez-vous et suppression des anciens

Dans les chapitres précédents, nous avons implémenter l'ensemble des méthodes représentant le trait **BDD**. Ces méthodes sont fréquemment utilisées quelque soit les bases de données à implémenter. Par contre, il est possible de rajouter des fonctions annexes pour réaliser d'autres requêtes spécifiques sachant que la connexion à une base de données se fait en dehors de la structure **RendezVous**.

*Pour la suite du projet, nous allons implémenter deux nouvelles fonctions. La première fonction **supprimer_rdv_obsoletes()** permet d'enlever tous les rendez-vous dont la date est dépassée. La deuxième **liste()** nous donne l'ensemble des rendez-vous enregistrés en donnant en premier les rendez-vous les plus urgents.*

*Pour la deuxième fonction, j'utilise une nouvelle méthode de **Connection** nommée **query_map()**. Cette méthode est similaire à **query_row()** avec exactement la même syntaxe d'utilisation interne. La seule différence est juste le résultat obtenu. La méthode **query_row()** donne une seule ligne d'enregistrement alors que la méthode **query_map()** renvoie l'ensemble des lignes concernées par la requête (**collection**).*

main.rs

```rust
mod bdd;
mod rdv_bdd;

use chrono::prelude::*;
use rendez_vous::{RendezVous, saisie};
use bdd::*;
use rusqlite::{params, Connection, Result, named_params};

fn main() {
  let mut rdv = RendezVous::now();
  let bdd = rdv.ouverture("rdv.db3");
  for rdv in liste(&bdd) {
    print!("Avant : ({id}) ", id=rdv.id);
    rdv.affiche();
  }
  println!("Aujourd'hui : {}", Local::now());
  supprimer_rdv_obsoletes(&bdd);
  for rdv in liste(&bdd) {
    print!("Après : ({id}) ", id=rdv.id);
```

```
    rdv.affiche();
  }
}

fn liste(bdd: &Connection) -> Vec<RendezVous> {
  let mut requete = bdd.prepare("SELECT * FROM RDV ORDER BY date").unwrap();
  let lignes = requete.query_map([], |ligne| {
    let jour : String = ligne.get(2)?;
    Ok(RendezVous {
      id: ligne.get(0)?,
      titre: ligne.get(1)?,
      date: jour.parse::<DateTime<Local>>().unwrap()
    })
  }).unwrap();
  let mut enregistrements = vec![];
  for ligne in lignes {
    if let Ok(enregistrement) = ligne {
      enregistrements.push(enregistrement);
    }
  }
  enregistrements
}

fn supprimer_rdv_obsoletes(bdd: &Connection) {
  bdd.execute("DELETE FROM RDV WHERE date < CURRENT_TIMESTAMP", []);
}
```

*Le développement de la **clôture** associée à la méthode **query_map()** possède exactement le même codage que la méthode **query_row()**. Cette fois-ci le résultat donne un ensemble de lignes et non plus un seul enregistrement.*

Résultat

```
Ouverture de la base 'rdv.db3'
Avant : (5) Football le samedi 27 février 2021 à 09h50mn
Avant : (4) Dentiste le samedi 11 septembre 2021 à 15h30mn
Avant : (1) Ophtalmologue le samedi 18 décembre 2021 à 15h00mn
Avant : (3) Réunion pédagogique le dimanche 23 janvier 2022 à 08h00mn
Avant : (6) Danse le mardi 15 février 2022 à 11h20mn
Aujourd'hui : 2021-09-21 17:08:53.316412912 +02:00
Après : (1) Ophtalmologue le samedi 18 décembre 2021 à 15h00mn
Après : (3) Réunion pédagogique le dimanche 23 janvier 2022 à 08h00mn
Après : (6) Danse le mardi 15 février 2022 à 11h20mn
```