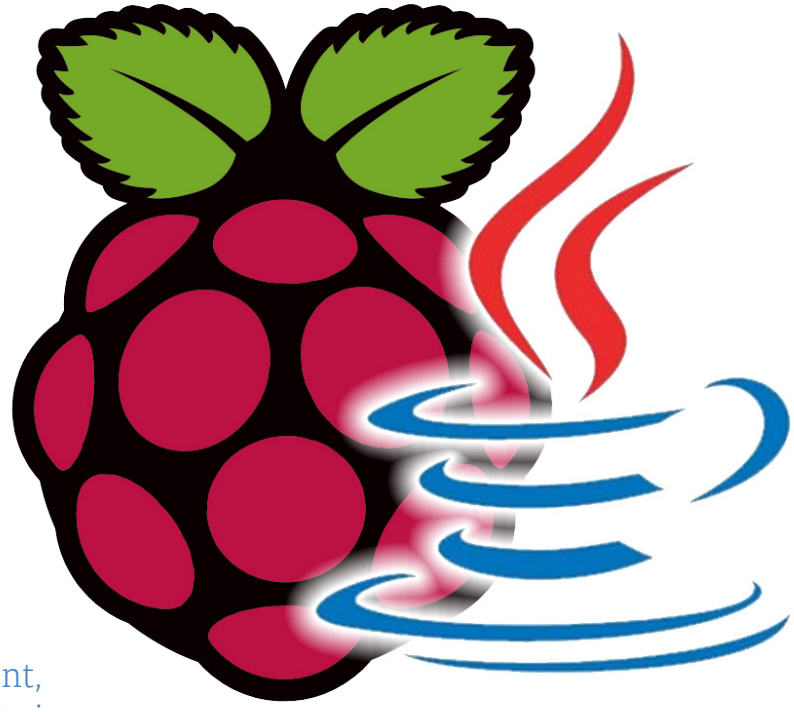
# Java sur Raspberry Pi

## Partie 2 : commande des broches GPIO avec un service REST de Spring



**Frank Delporte** (Belgique)

Java est-il un langage de programmation adapté au Raspberry Pi ? Dans l'article précédent, nous avons répondu à cette question par un « oui » retentissant ! Avec quelques applications de démonstration Java à fichier unique à notre actif, il est maintenant temps d'approfondir. En développant ce que nous avons couvert, nous irons maintenant plus loin et construirons une application complète avec plusieurs classes. Celles-ci vont nous fournir des services web REST pour commander les broches GPIO d'un Raspberry Pi.

La version « Raspberry Pi OS Full (32 bits) » du système d'exploitation comprend déjà la version 11 du kit de développement Java (JDK). Cependant, pour développer des applications Java, un EDI nous aidera à écrire des applications faciles à maintenir. Comme indiqué dans l'article précédent [1], Visual Studio Code peut être utilisé sur le RPi. L'autre solution est de programmer l'application dans Visual Studio Code sur un PC, puis de la compiler et l'exécuter sur le RPi. Pour cet article, nous écrirons le code directement sur le RPi. Pour cela, nous avons besoin de quelques outils supplémentaires, alors commençons par les installer.

### Maven

Nous nous servirons de Maven pour construire l'application sur notre RPi. Maven compile le code, ainsi que les dépendances requises, dans un seul fichier JAR. Ceci est possible grâce au fichier de configuration *pom.xml* qui se trouve à la racine du projet.
Une seule commande suffit pour installer Maven, après quoi nous pouvons immédiatement vérifier l'installation en demandant la version comme suit :

```
$ sudo apt install maven
$ mvn -v
Apache Maven 3.6.0
Maven home: /usr/share/maven
```

### Pi4J

Pour commander les broches GPIO, nous aurons recours à la bibliothèque Pi4J qui établit un pont entre notre code Java et les broches d'entrée/sortie à usage général (GPIO) du RPi. Cela nous permet de commander les broches GPIO et de nous connecter à divers composants électroniques.
Pour une prise en charge complète de la bibliothèque Pi4J sur le RPi, nous devons installer quelques logiciels supplémentaires. Une nouvelle fois, une seule commande suffit pour le faire :

```
$ curl -sSL https://pi4j.com/install | sudo bash
```

### Mise à jour de WiringPi

Une dernière étape est nécessaire pour être fin prêt. Si vous utilisez un RPi 4, vous devrez mettre à jour WiringPi. Pi4J l'utilise comme bibliothèque native pour commander les broches GPIO. Comme l'architecture du système sur puce (SoC) a changé avec la version 4, il vous faut la nouvelle version de WiringPi. Malheureusement, ce projet est obsolète depuis l'année dernière, mais grâce à la communauté *open source*, une version « non officielle » est disponible et peut être installée par un script fourni par Pi4J. Exécutez la commande :

```
sudo pi4j -wiringpi
```

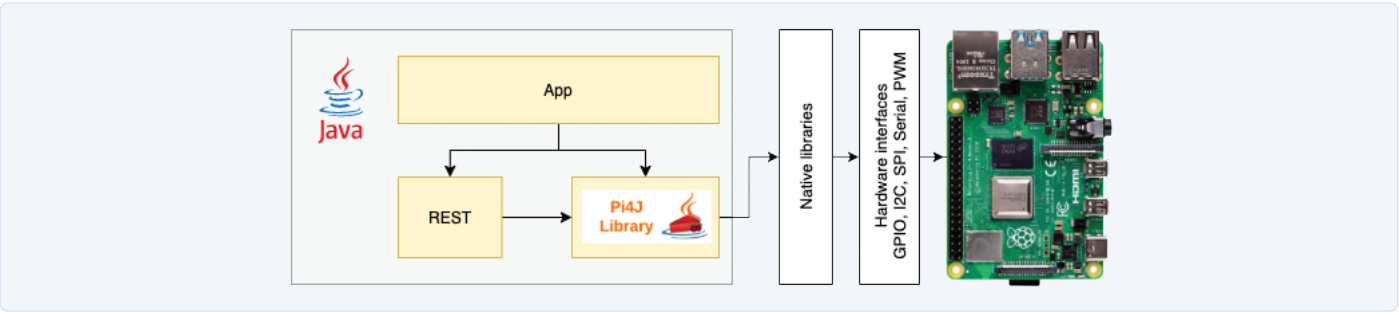



*Figure 1. Aperçu de l'application utilisant le service REST de Spring et la bibliothèque Pi4J.*

Elle récupère le projet depuis GitHub, le compile et l'installe sur votre RPi d'un seul coup. En demandant la version, vous verrez qu'elle a été mise à niveau de la version par défaut, de 2.50 à 2.60 :

```
$ gpio -v
gpio version: 2.50
$ sudo pi4j -wiringpi
$ pi4j -v
------------------------------------------
              THE Pi4J PROJECT
------------------------------------------
  PI4J.VERSION     : 1.3
  PI4J.TIMESTAMP   : 2021-01-28 04:14:07
------------------------------------------
  WIRINGPI.PATH : /usr/lib/libwiringPi.so /usr/local/lib/
    libwiringPi.so
  WIRINGPI.VERSION : 2.60
------------------------------------------
```


*Figure 2. Paramètres de configuration requis pour Spring Initializr.*

## L'application

Le code complet de cette application est disponible sur le dépôt GitHub qui accompagne cet article [2] dans le dossier `elektor/2106`. Ce projet est une démonstration de faisabilité qui commande les broches GPIO en utilisant un service web REST. Il utilise le canevas Spring, un logiciel qui fournit de nombreux outils pour construire des applications puissantes avec un minimum de code (**fig. 1**). Nous décrivons dans ce qui suit le processus de création des différents fichiers qui composent ce projet. Si cela ne fonctionne pas comme décrit, le code du dépôt devrait fonctionner tel quel.
Depuis GitHub, on récupère le projet complet ainsi :

```
pi@raspberrypi:~ $ git clone https://github.com/FDelporte/
    elektor
Cloning into 'elektor'...
remote: Enumerating objects: 34, done.
remote: Counting objects: 100% (34/34), done.
remote: Compressing objects: 100% (24/24), done.
remote: Total 34 (delta 2), reused 34 (delta 2), pack-reused 0
Unpacking objects: 100% (34/34), done.
pi@raspberrypi:~ $ cd elektor/2106
pi@raspberrypi:~/elektor/2106 $ ls -l
total 8
-rw-r--r-- 1 pi pi 1720 Feb 15 14:23 pom.xml
drwxr-xr-x 3 pi pi 4096 Feb 15 14:23 src
```

## Qu'est-ce que Spring et ses outils ?

Spring est un canevas qui simplifie et accélère le développement d'applications Java (commerciales). Spring Boot est une couche qui se trouve au-dessus de Spring et qui fournit des paquets « prêts à l'emploi ». Ceux-ci permettent de créer des applications autonomes basées sur Spring que vous pouvez « exécuter simplement ». Ceci est rendu possible grâce à un principe connu sous le nom de « convention plutôt que configuration ». Cela signifie que, par défaut, tout fonctionne selon une convention prédéfinie. Si vous avez besoin de faire quelque chose de différent, vous pouvez alors modifier la configuration. Les possibilités les plus importantes, telles que répertoriées sur le site Web de Spring Boot [3], sont les suivantes :

- Création facile d'applications Spring autonomes.
- Intégration d'un serveur web dans votre application (Tomcat, Jetty ou Undertow).
- Fourniture de dépendances de type « *starter* » pour simplifier votre configuration de *build*.
- Configuration automatique de Spring et des bibliothèques tierces lorsque cela est possible.
- Fourniture de fonctions prêtes pour la production, telles que des indicateurs des bilans de santé et la configuration externalisée.
- Absolument aucun besoin de génération de code ou de configuration XML.

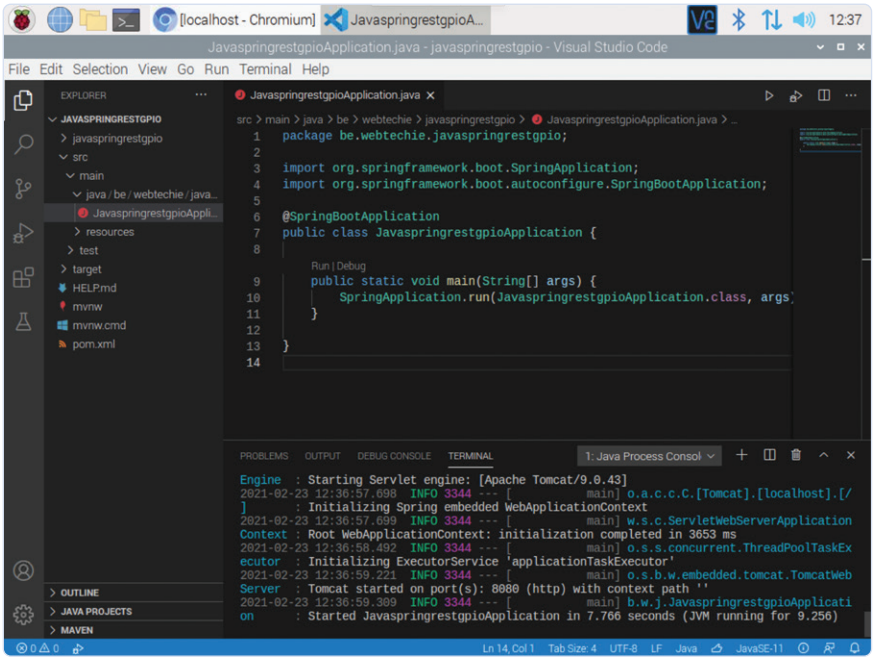



*Figure 3. Exécution du projet dans Visual Studio Code.*

Enfin, il existe Spring Initializr [4], un outil en ligne permettant de créer rapidement une application de démarrage comprenant tous les paquets Spring Boot requis.

## Pour commencer

Pour créer ce projet à partir de zéro, nous commençons par utiliser Spring Initializr, un formulaire de configuration en ligne, qui créera un package de démarrage pour nous. Suivez le lien [4] et recopiez les options dans le formulaire en utilisant les informations de la capture d'écran de la **figure 2**. Sur le côté droit, assurez-vous également de cliquer sur le bouton « *Dependencies* ». Dans la fenêtre qui s'ouvre, nous devons ajouter la dépendance « Spring Web ». On peut la trouver rapidement en utilisant la boîte de recherche. En cliquant sur cette entrée, on l'ajoute au projet. Si une dépendance non désirée est ajoutée par erreur, il suffit de cliquer sur le bouton rouge (tout à droite) pour la supprimer. L'ajout de la dépendance « Spring Web » permet de construire une application REST [5], nous pourrons ainsi nous connecter à l'application via des pages Web dans notre navigateur.

En cliquant sur le bouton « *Generate* », on obtient un fichier ZIP contenant un projet Maven prêt à l'emploi. Une fois qu'il a été décompressé dans un répertoire approprié, ouvrez-le dans Visual Studio Code [6]. Cela fonctionne, à ce stade, aussi bien sur le RPi que sur un PC. Utilisez simplement *Fichier -> Ouvrir le dossier...* et sélectionnez le dossier décompressé. Ensuite, trouvez le fichier *JavaSpringRestApplication.java* (sous *src/main/java*). En cliquant avec le bouton droit de la souris sur ce nom de fichier, l'option *Run* est disponible. Sélectionnez *Run* et l'application devrait démarrer (**fig. 3**). Si votre installation Visual Studio Code ne dispose pas déjà du support Java requis, une fenêtre pop-up apparaîtra pour demander son installation.

Dans le panneau Terminal, nous obtenons le résultat du **listage 1** (ici sans horodatage). Que nous apprend cette sortie ?

- La JVM (Java Virtual Machine) a fonctionné pendant 7 s, dont 4,5 s pour lancer l'application.
- Tomcat (le serveur web) est intégré à l'application et fonctionne sur le port 8080.

**Listage 1. Sortie d'une application starter.**

```
°  .°°  ____  °°°°°°°°°°  _°°°°°°°°°°°  __ _ _
°/\\ / ___'_ __ _ _(_)_ __°  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
°\\/° ___)| |_)| | | | | || (_| |° ) ) ) )
°  '°  |____| .__|_| |_|_| |_\__, | / / / /
°=========|_|==============|___/=/_/_/_/
°:: Spring Boot ::°°°°°°°  (v2.4.2)
°
Starting JavaSpringRestApplication
No active profile set, falling back to default profiles: default
Tomcat initialized with port(s): 8080 (http)
Starting service [Tomcat]
Starting Servlet engine: [Apache Tomcat/9.0.41]
Initializing Spring embedded WebApplicationContext
Root WebApplicationContext: initialization completed in 1607 ms
Initializing ExecutorService 'applicationTaskExecutor'
Tomcat started on port(s): 8080 (http) with context path ''
Started JavaSpringRestApplication in 4.582 seconds (JVM running for 7.196)
Initializing Spring DispatcherServlet 'dispatcherServlet'
Initializing Servlet 'dispatcherServlet'
Completed initialization in 9 ms
```

Joli, alors ouvrons un navigateur et allons sur `http://localhost:8080/` (**fig. 4**).

Nous obtenons une page d'erreur – un peu décevant. En effet, nous n'avons encore rien créé qui puisse être affiché sur cette page ! Cependant, cette page d'erreur prouve que notre application fonctionne et se comporte comme un serveur web. Elle ne sait simplement pas encore ce qu'elle doit partager avec nous..

## Construction de l'application

Maintenant, étendons ce package de démarrage Spring pour inclure la fonction voulue dans notre application, à savoir la commande des broches GPIO avec du code Java. Pour cela, nous devons inclure la bibliothèque Pi4J dans notre projet. Grâce à elle, nous n'avons pas besoin de commander le matériel nous-mêmes, car nous pouvons utiliser l'ensemble des méthodes Java fournies par Pi4J. Cependant, n'oubliez pas que cette bibliothèque dépend de matériel présent sur le RPi, donc nous ne pouvons pas exécuter ou tester entièrement cette application sur le PC.

Le fichier JAR produit devra être exécuté sur le RPi pour tester la commande des broches GPIO.
Dans les exemples de code fournis ici, chaque variable ou méthode est documentée dans le style JavaDoc (en commençant par `/**`) pour expliquer son but.

## Ajout des dépendances

Avec le projet créé avec Spring Initializr ouvert, nous ajoutons la dépendance `pi4j-core` au fichier *pom.xml*. Cela garantit que nous pouvons accéder aux méthodes pi4j :

```
<dependency>
<groupId>com.pi4j</groupId>
<artifactId>pi4j-core</artifactId>
<version>1.3</version>
<scope>compile</scope>
</dependency>
```

Pendant que nous y sommes, nous pouvons également ajouter la dépendance OpenAPI (`springdoc-openapi-ui`) que nous utiliserons plus tard pour tester les services REST :

```
<dependency>
<groupId>org.springdoc</groupId>
<artifactId>springdoc-openapi-ui</artifactId>
<version>1.5.1</version>
</dependency>
```

## Ajout d'un contrôleur d'information REST

D'abord nous souhaitons que l'application nous donne les informations sur le RPi accessibles avec la bibliothèque Pi4J. Nous commençons par créer un package « *controller* » avec un fichier *InfoRestController.java*. Dans Visual Studio Code, cliquez avec le bouton droit de la souris sur l'entrée *java\be\webtechie\javaspringrestgpio* et sélectionnez *Nouveau dossier*. Nommez le nouveau dossier « *controller* ». Ensuite, cliquez avec le bouton droit de la souris sur le dossier *controller* et sélectionnez *Nouveau fichier.* Nommez le fichier *InfoRestController.java*.
Dans les sources, vous trouverez le code complet, mais le **listage 2** fournit un court extrait de son contenu. Chaque méthode est une projection REST qui renvoie un ensemble spécifique de paires clé-valeur contenant des informations sur le RPi.

## Ajout du gestionnaire GPIO

Avant de pouvoir créer le contrôleur GPIO REST, nous ajoutons un fichier *GpioManager.java* pour gérer les appels Pi4J. Ce fichier est placé dans un package « *manager* » (créé comme un *nouveau dossier* et un *nouveau fichier* comme précédemment). Nous utiliserons ce gestionnaire pour stocker les broches GPIO initialisées et appeler les méthodes Pi4J pour interagir avec les broches GPIO. Encore une fois, un court extrait du code est fourni dans le **listage 3**,

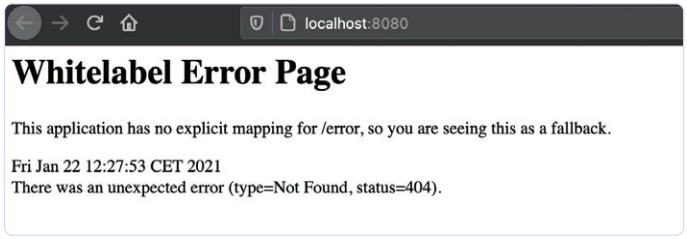



*Figure 4. La page web d'erreur par défaut prouve que le serveur web est opérationnel.*

**Listage 2. Extrait du fichier InfoRestController.java.**

```
/**
 * Provides a REST-interface to expose all board info.
 */
@RestController
@RequestMapping("info")
public class InfoRestController {
    private Logger logger = LoggerFactory.getLogger(this.getClass());

    /**
     * Get the OS info.
     */
    @GetMapping(path = "os", produces = "application/json")
    public Map<String, String> getOsInfo() {
        Map<String, String> map = new TreeMap<>();
        try {
            map.put("Name", SystemInfo.getOsName());
        } catch (Exception ex) {
            logger.error("OS name not available, error: {}",
                    ex.getMessage());
        }
        try {
            map.put("Version", SystemInfo.getOsVersion());
        } catch (Exception ex) {
            logger.error("OS version not available, error: {}",
                    ex.getMessage());
        }
        return map;
    }

    /**
     * Get the Java info.
     */
    @GetMapping(path = "java", produces = "application/json")
    public Map<String, String> getJavaInfo() {
        Map<String, String> map = new TreeMap<>();
        map.put("Vendor ", SystemInfo.getJavaVendor());
        map.put("VendorURL", SystemInfo.getJavaVendorUrl());
        map.put("Version", SystemInfo.getJavaVersion());
        map.put("VM", SystemInfo.getJavaVirtualMachine());
        map.put("Runtime", SystemInfo.getJavaRuntime());
        return map;
    }

    ...
}
```

**Listage 3. Extrait du fichier GpioManager.java.**

```
/**
 * Service instance managing the {@link GpioFactory}.
 */
@Service
public class GpioManager {
    private Logger logger = LoggerFactory.getLogger(this.getClass());

    /**
     * The GPIO controller.
     */
    private final GpioController gpio;

    /**
     * List of the provisioned pins with the address as key.
     */
    private final Map<Integer, Object> provisionedPins =
            new HashMap<>();

    /**
     * Constructor which initializes the Pi4J {@link GpioController}.
     */
    public GpioManager() {
         String osName = SystemInfo.getOsName();
         if (osName.toLowerCase().contains("raspberry") || osName.toLowerCase().contains("linux")) {
             this.gpio = GpioFactory.getInstance();
         } else {
             logger.error("GPIO could not be initialized. Not running on Raspberry Pi but '{}'", osName);
             this.gpio = null;
         }
     }

    /**
     * Get the pin for the given address.
     *
     * @param address The address of the GPIO pin.
     * @return The {@link Pin} or null when not found.
     */
    private Pin getPinByAddress(int address) {
        Pin pin = RaspiPin.getPinByAddress(address);
        if (pin == null) {
            logger.error("No pin available for address {}", address);
        }
        return pin;
    }

    /**
     * Provision a GPIO as digital output pin.
     *
     * @param address The address of the GPIO pin.
```

tandis que le code complet de cette classe se trouve dans les sources du dépôt. Notez que nous utilisons `@Service` dans cette classe qui indique au canevas Spring de garder une seule instance de cet objet en mémoire. Cela garantit que nous maintenons en permanence une cartographie des broches GPIO sélectionnées.

## Ajout du contrôleur GPIO REST

Enfin, nous ajoutons un contrôleur GPIO avec une interface REST au package (dossier) du *contrôleur*. Ce fichier nommé *GpioRestController.java* expose les méthodes GPIO de Pi4J définies dans la classe *GpioManager.java*. Une fois encore, un extrait du code est présenté dans le **listage 4** et le code complet est disponible dans le dépôt.

## Exécution de l'application

Cette étape peut être réalisée à la fois sur le PC et sur le RPi. Sur le PC, il est préférable d'exécuter l'application à partir de Visual Studio Code. Cela évite d'avoir à installer Maven sur le PC.

Comme nous avons ajouté la dépendance *springdoc-openapi-ui* dans le fichier *pom.xml*, l'application nous fournira une page web Swagger très utile pour tester les services REST. Swagger est un autre projet *open source* qui crée une interface de page web simple pour tester notre code Java en visualisant automatiquement les contrôleurs que nous avons créés.

Il existe deux façons de lancer l'application. Dans Visual Studio Code, vous la lancez avec *Run* (RPi ou PC). L'alternative est de construire l'application dans un fichier jar avec *mvn package* (RPi uniquement). Une fois

```
     * @param name The name of the GPIO pin.
     * @return True if successful.
     */
    public boolean provisionDigitalOutputPin(final int address,
            final String name) {
        if (this.provisionedPins.containsKey(address)) {
            throw new IllegalArgumentException("There is already"
                  + " a provisioned pin at the given address");
        }

        final GpioPinDigitalOutput provisionedPin = this.gpio
            .provisionDigitalOutputPin(
                this.getPinByAddress(address), name, PinState.HIGH);
        provisionedPin.setShutdownOptions(true, PinState.LOW);

        this.provisionedPins.put(address, provisionedPin);

        return true;
    }

    /**
     * Toggle a pin.
     *
     * @param address The address of the GPIO pin.
     * @return True if successful.
     */
    public boolean togglePin(final int address) {
        logger.info("Toggle pin requested for address {}", address);

        Object provisionedPin = this.provisionedPins.get(address);

        if (provisionedPin == null) {
            throw new IllegalArgumentException("There is no pin"
                + " provisioned at the given address");
        } else {
            if (provisionedPin instanceof GpioPinDigitalOutput) {
                ((GpioPinDigitalOutput) provisionedPin).toggle();

                return true;
            } else {
                throw new IllegalArgumentException("The provisioned pin"
                        + " at the given address is not of the type"
                        + " GpioPinDigitalOutput");
            }
        }
    }

    ...

}
```


*Figure 5. La page Swagger permet d'accéder aux API REST du contrôleur info et GPIO.*

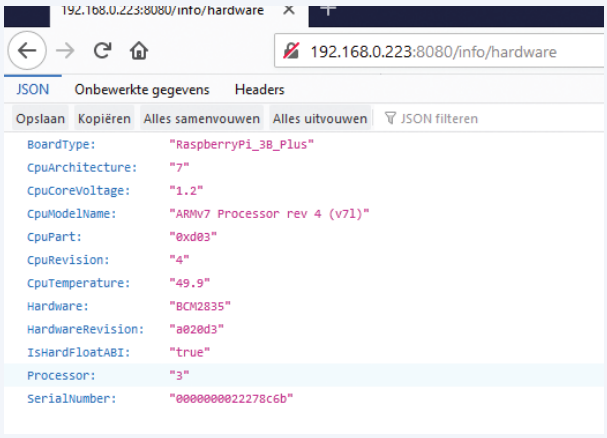
**Listage 4. Extrait du fichier GpioRestController.java.**

```
/**
 * Provides a REST-interface to interact with the GPIOs.
 */
@RestController
@RequestMapping(“gpio”)
public class GpioRestController {
    private Logger logger = LoggerFactory.getLogger(this.getClass());

    /**
     * Reference to the GPIO manager service
     */
    private final GpioManager gpioManager;

    /**
     * Constructor used by Spring to “inject” the GPIO manager
     * into this class.
     *
     * @param gpioManager {@link GpioManager}
     */
    public GpioRestController(GpioManager gpioManager) {
        this.gpioManager = gpioManager;
    }

    ...

    /**
     * Provision a GPIO as digital output pin.
     *
     * @param address The address of the GPIO pin.
     * @param name The name of the GPIO pin.
```

construite, elle est exécutée comme suit à partir de la ligne de commande :

```
java -jar target/javaspringrestgpio-0.0.1-
    SNAPSHOT.jar
```

Dans un navigateur sur le RPi, ouvrez la page Swagger en utilisant `http://localhost:8080/swagger-ui.html`. Autre solution : depuis n'importe quel PC sur le même réseau, utilisez `http://<IP_ADDRESS_RASPBERRY_PI>:8080/swagger-ui.html`. Les deux contrôleurs avec leurs méthodes seront affichés comme à la **figure 5**.



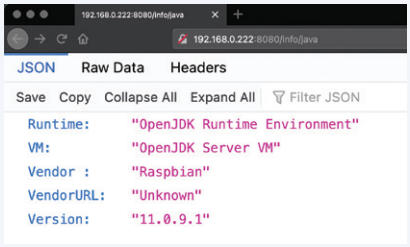




*Figure 6. Les données JSON peuvent être acquises directement en ajoutant le nom de la méthode à l'URL.*

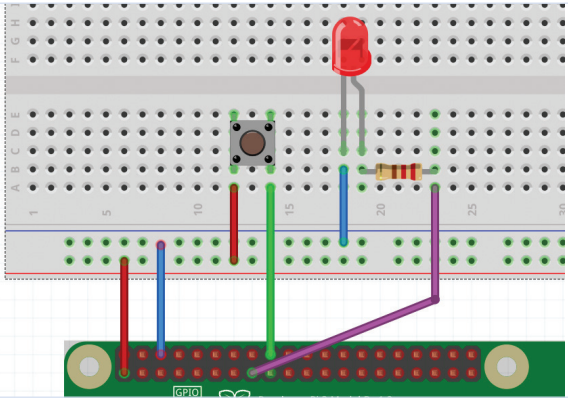


*Figure 7. Schéma de câblage pour connecter l'interrupteur et la LED au Raspberry Pi.*

## Test du contrôleur d'information REST

Nous pouvons cliquer sur les boutons de la page Swagger et exécuter les options disponibles. Par exemple, dans la section *info-rest-controller*, cliquez sur *GET* à côté de la méthode *info/hardware*. Ensuite cliquez sur *Try it out* puis sur *Execute* qui affiche la réponse dans la section *Response body*.

Toutes ces méthodes peuvent également être appelées directement depuis le navigateur. Il suffit d'ajouter le nom de la méthode à l'URL. Pour *info/hardware*, il suffit d'entrer `http://localhost:8080/info/hardware`, et pour *info/java*, c'est `http://localhost:8080/info/java`. Les données issues de l'appel de ces méthodes sont affichées au format JSON, comme le montre la **figure 6**.

```
     * @return True if successful.
     */
    @PostMapping(
        path = “provision/digital/output//”,
        produces = “application/json”)
    public boolean provisionDigitalOutputPin(
            @PathVariable(“address”) int address,
            @PathVariable(“name”) String name) {
        return this.gpioManager
            .provisionDigitalOutputPin(address, name);
    }

    ...

    /**
     * Toggle a pin.
     *
     * @param address The address of the GPIO pin.
     * @return True if successful.
     */
    @PostMapping(
        path = “digital/toggle/”,
        produces = “application/json”)
    public boolean togglePin(@PathVariable(“address”) long address) {
        return this.gpioManager.togglePin((int) address);
    }

    ...
}
```

## Tester le contrôleur GPIO REST avec une LED et un bouton

Pour montrer comment cette application peut interagir avec les broches GPIO, un montage très simple sur plaque d’essai suffira (**fig. 7**) :

- LED sur la broche GPIO 5, BCM 22, WiringPi n°3
- Bouton sur la broche GPIO 18, BCM 24, WiringPi n°5

Avant de commander la LED connectée et de lire l’état du bouton, il faut initialiser les broches

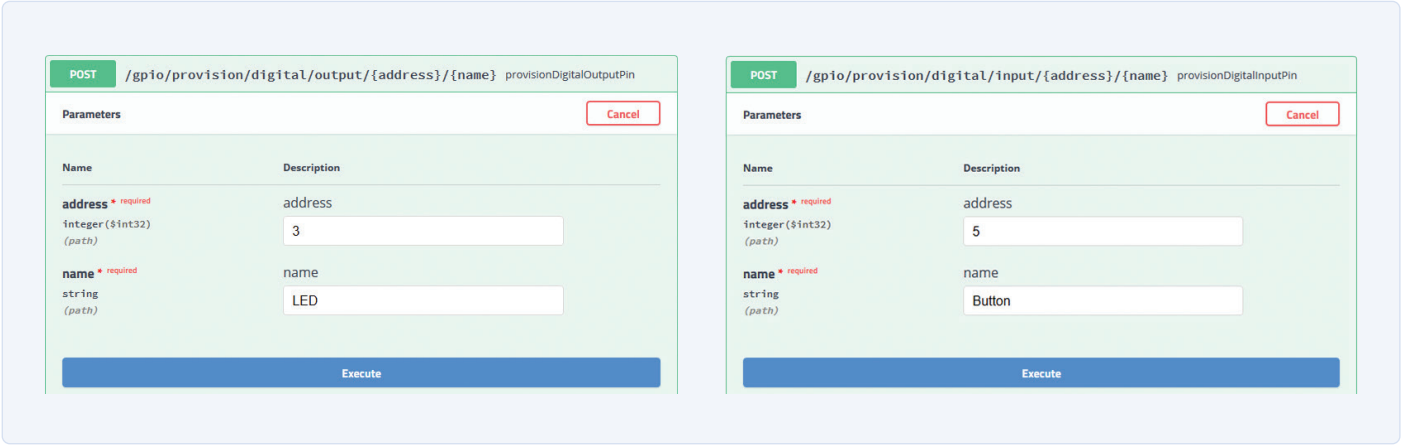



*Figure 8. Initialisation des GPIO en utilisant les méthodes* `/gpio/provision/digital/`.

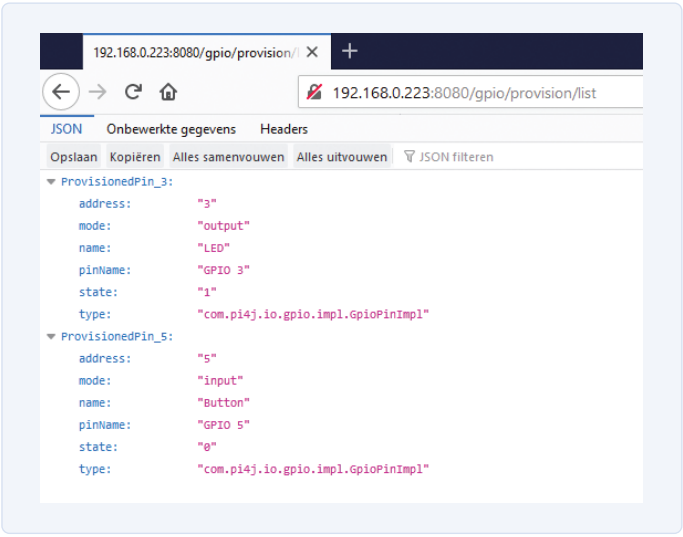



*Figure 9. Demande de la configuration GPIO avec la méthode* `/gpio/provision/list`.

GPIO. La méthode `/gpio/provision/digital/output` permet de configurer une sortie. De retour à la page `http://localhost:8080/swagger-ui.html`, cliquez sur *GET by this method*, *Try it out* puis entrez *3* dans *address* et *LED* dans *string*. Validez la configuration en cliquant sur *Execute* (**fig. 8**, à gauche).

La méthode `/gpio/provision/digital/input` permet de configurer une entrée. Cliquez sur *GET by this method*, *Try it out*, puis entrez *5* dans *address* et *Button* dans *string*. Validez la configuration en cliquant sur *Execute* (**fig. 8**, à droite).

Une fois les broches GPIO initialisés, la méthode `/gpio/provision/list` permet d'en obtenir la liste. Il suffit d'utiliser *GET*, *Try it out*, et *Execute* ou de taper l'URL `http://localhost:8080/gpio/provision/list` directement dans le navigateur (**fig. 9**).

Maintenant que nous avons vérifié que les broches GPIO sont prêtes à être utilisées, nous pouvons allumer et éteindre la LED en utilisant la méthode `/gpio/digital/toggle` en cliquant plusieurs fois sur le bouton *Execute* (**fig. 10**).

Il existe également une méthode supplémentaire qui permet d'allumer la LED pendant une durée donnée, par ex. 2 s (**fig. 11**). Nota : la durée doit être fournie en millisecondes.

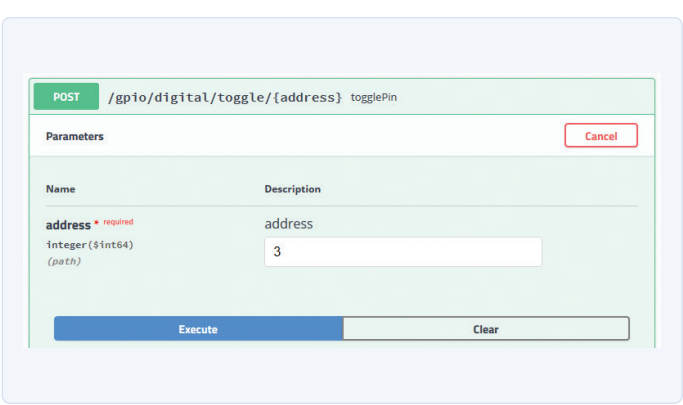



*Figure 10. Utilisation du bouton Execute dans l'interface Swagger pour faire basculer la LED.*

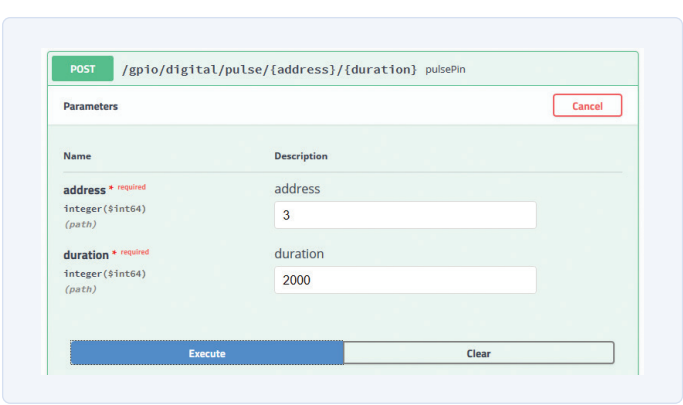



*Figure 11. La LED peut également être activée pendant une durée déterminée.*

L'interface Swagger permet également de consulter l'état du bouton, également disponible directement via l'URL `http://localhost:8080/gpio/state/5`. Dans le cas de la **figure 12,** le bouton est pressé et renvoie un 1.

## Poursuite de l'exploration

Cette application ne présente que quelques-unes des méthodes Pi4J en tant que services REST, cela montre les possibilités et la puissance de cette approche. En fonction de votre projet, vous pouvez étendre ou retravailler cet exemple pour qu'il réponde à vos besoins.
La bibliothèque Pi4J est en cours de réécriture afin d'offrir une meilleure prise en charge du RPi 4 et des futures versions. Cela permettra également de la mettre à jour avec les dernières versions de Java. En attendant, la version actuelle vous permet de démarrer et de créer des applications dans lesquelles vous intégrerez le contrôle du matériel via une API REST.

(200617-B-04)

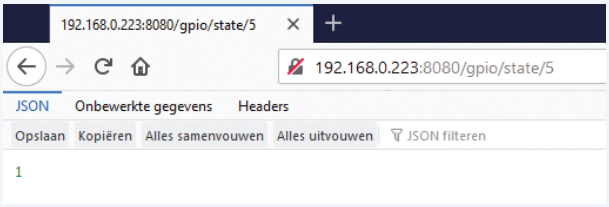



*Figure 12. L'URL avec la méthode `gpio/state/5` acquiert l'état du bouton.*

**Contributeurs**
Idée, texte et images : **Frank Delporte**
Rédaction : **Stuart Cording**
Mise en page : **Giel Dols**
Traduction : **Denis Lafourcade**

**Des questions, des commentaires ?**
Envoyez un courriel à l'auteur (javaonraspberrypi@webtechie.be) ou contactez Elektor (redaction@elektor.fr).



## LIENS

[1] **F. Delporte, « Java sur le Raspberry Pi – partie 1 : les broches GPIO », Elektor, 05-06/2021** : www.elektormagazine.fr/200617-04
[2] **Dépôt GitHub pour cet article** : https://bit.ly/3i9bP4v
[3] **Documentation de Spring Boot** : https://spring.io/projects/spring-boot
[4] **Spring Initialzr** : https://start.spring.io/
[5] **« What Is REST? », Codecademy** : https://bit.ly/31odThv
[6] **Visual Studio Code** : https://code.visualstudio.com/