howest
hogeschool

Object Oriented Architectures and Secure Development

Configuration Files

Arne Debou
Mattias De Wael
Frédéric Vlummens

howest hogeschool

Introducing configuration files

Part 1

What are configuration files and why use them?

- Configuration files can be used to configure parameters and initial settings for your application.
- For example:
  - Database connection details
  - SMTP server to use
  - …
- We prefer configuration files over hardcoding this kind of information in our application’s .java files.
  - We don’t need to recompile the application if a parameter changes.
  - Configuration files can be stored anywhere we want.
  - Change application’s behavior.

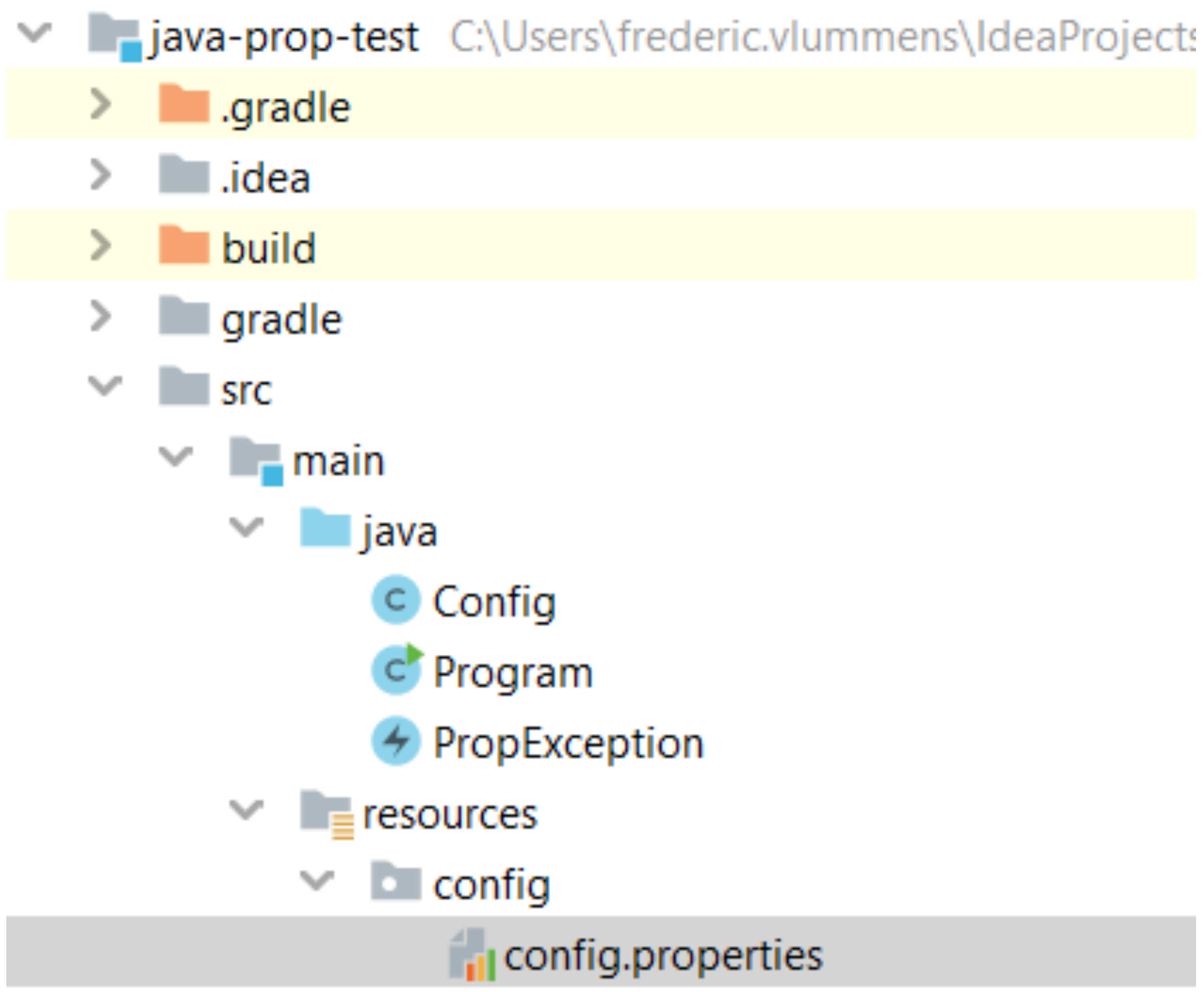
Configuration files in Java

- You could implement your own mechanism files…
- However, built-in mechanism exists, using .properties files
- .properties files are text files, containing key-value pairs
- Example:

```
name=Frédéric
age=43
```

Where to store the .properties file?

- Can be stored in **/resources**
- Convention in this class:
  /resources/config/config.properties
- Notes:
  - You can have multiple .properties files (one for database config, one for mail server config, …)
  - You may store your .properties files elsewhere
  - It all depends on the situation…



```
java-prop-test  C:\Users\frederic.vlummens\IdeaProjects
  .gradle
  .idea
  build
  gradle
  src
    main
      java
        Config
        Program
        PropException
      resources
        config
          config.properties
```

Reading from a .properties file

1)
```java
Properties properties = new Properties();
```

2)
```java
try (InputStream ris = getClass().getResourceAsStream(CONFIG_FILE)) {
    properties.load(ris);
} catch (IOException ex) {
    LOGGER.log(Level.SEVERE,
            "Unable to read config file", ex);
    throw new HowestException("Unable to load configuration.");
}
```

3)
```java
String name = properties.getProperty("name");
```

Step 1: initialize properties object
Step 2: load the properties from the file
Step 3: retrieve a property from the properties object

Updating a property and writing to file

1)
```
properties.setProperty("name", "Mattias");
```

2)
```
String path = getClass().getResource(CONFIG_FILE).getPath();

try (FileOutputStream fos = new FileOutputStream(path)) {
    properties.store(fos, null);
} catch (IOException ex) {
    LOGGER.log(Level.SEVERE,
            "Unable to write config file", ex);
    throw new PropException("Unable to save configuration.");
}
```

Step 1: change the property
Step 2: write the properties back to file

Writing to .properties file – attention point: src vs build

- When compiling, your resources (including .properties files) are copied from **/src/resources/** to **/build/resources/**
- When you update a property using code at run-time, only the file in **/build/resources/** is updated accordingly.
- Therefore, it is perfectly logical that the config file in **/src/resources/** remains the same.
- And at next run, it will once again be copied from **/src/resources** to **/build/resources**!
- This problem does not occur once you deploy your application.

.properties files: reusing code

- Up til now, we wrote .properties manipulation code in our GUI layer itself…
- Let’s encapsulate this in a **Config** class.
- The **Config** class will be responsible for all reading and writing from/towards the config file.

Introducing the Config utility class

```java
public class Config {

    private static final String CONFIG_FILE = "/config/config.properties";
    private static final Config INSTANCE = new Config();
    private final Properties properties = new Properties();
    private static final Logger LOGGER = Logger.getLogger(Config.class.getName())

    private Config() {
        try (InputStream ris = getClass().getResourceAsStream(CONFIG_FILE)) {
            properties.load(ris);
        } catch (IOException ex) {
            LOGGER.log(Level.SEVERE,
                    "Unable to read config file", ex);
            throw new PropException("Unable to load configuration.");
        }
    }

    public static Config getInstance() {
        return INSTANCE;
    }
```

Introducing the Config utility class

```
public class Config {

    private static final String CONFIG_FILE = "/config/config.properties";
    private static final Config INSTANCE = new Config();
    private Properties properties = new Properties();
    private static final Logger LOGGER = Logger.getLogger(Config.class.getName())

    private Config() {
        try (InputStream ris = getClass().getResourceAsStream(CONFIG_FILE)) {
            properties.load(ris);
        } catch (IOException ex) {
            LOGGER.log(Level.SEVERE,
                    "Unable to read config file", ex);
            throw new PropException("Unable to load configuration.");
        }
    }

    public static Config getInstance() {
        return INSTANCE;
    }
```

Singleton pattern

Introducing the Config class

```java
public String readSetting(String key, String defaultValue) {
    return properties.getProperty(key, defaultValue);
}

public String readSetting(String key) {
    return readSetting(key, null);
}

public void writeSetting(String key, String value) {
    properties.setProperty(key, value);
    storeSettingsToFile();
}
```

Introducing the Config class

```java
public String readSetting(String key, String defaultValue) {
    return properties.getProperty(key, defaultValue);
}

public String readSetting(String key) {
    return readSetting(key, null);
}

public void writeSetting(String key, String value) {
    properties.setProperty(key, value);
    storeSettingsToFile();
}
```

Default value will be returned if no value provided in the .properties file

Introducing the Config class

```
private void storeSettingsToFile() {
    String path = getClass().getResource(CONFIG_FILE).getPath();

    try (FileOutputStream fos = new FileOutputStream(path)) {
        properties.store(fos, null);
    } catch (IOException | NullPointerException ex) {
        LOGGER.log(Level.SEVERE,
                "Unable to write config file", ex);
        throw new PropException("Unable to save configuration.");
    }
}
```

Using the Config class

```java
private void run() {
    Config conf = Config.getInstance();

    System.out.println(conf.readSetting("name"));
    System.out.println(conf.readSetting("age"));

    conf.writeSetting("name", "Joske");
}
```

- All logic for reading and writing to the file is nicely encapsulated in Config class.
- We don’t need to care about the details in the rest of our application.

howest
hogeschool

Storing database connection details in a configuration file

Part 2

Storing database URL, username and password

- Up til now, we have done this as plain text String constants in a Java file:

```java
public class MySqlConnection {

    private static final String URL = "jdbc:mysql://localhost/howest-shop?serverTimezone=UTC";
    private static final String USERNAME = "howest-shop-user";
    private static final String PASSWORD = "howest-shop-password"; // NOSONAR

    private MySqlConnection() {
    }

    public static Connection getConnection() throws SQLException {
        return DriverManager.getConnection(URL, USERNAME, PASSWORD);  // NOSONAR
    }

}
```

Storing database URL, username and password

- Problems with this approach:
  - When parameters change (e.g. new database server address), we need to change our source code and recompile.
  - Anyone taking a look at the source code immediately knows our database username and password.
- Let’s try and fix both issues!

Fixing issue 1: taking config parameters out of Java code

```java
public class MySqlConnection {

    private static final String KEY_DB_URL = "db.url";
    private static final String KEY_DB_USERNAME = "db.username";
    private static final String KEY_DB_PASSWORD = "db.password"; // NOSONAR

    private static final String url;
    private static final String username;
    private static final String password;

    static {
        url = Config.getInstance().readSetting(KEY_DB_URL);
        username = Config.getInstance().readSetting(KEY_DB_USERNAME);
        password = Config.getInstance().readSetting(KEY_DB_PASSWORD);
    }

    private MySqlConnection() {
    }

    public static Connection getConnection() throws SQLException {
        return DriverManager.getConnection(url, username, password);
    }
```

Fixing issue 1: taking config parameters out of Java code

```java
public class MySqlConnection {

    private static final String KEY_DB_URL = "db.url";
    private static final String KEY_DB_USERNAME = "db.username";
    private static final String KEY_DB_PASSWORD = "db.password"; // NOSONAR

    private static final String url;
    private static final String username;
    private static final String password;

    static {
        url = Config.getInstance().readSetting(KEY_DB_URL);
        username = Config.getInstance().readSetting(KEY_DB_USERNAME);
        password = Config.getInstance().readSetting(KEY_DB_PASSWORD);
    }

    private MySqlConnection() {
    }

    public static Connection getConnection() throws SQLException {
        return DriverManager.getConnection(url, username, password);
    }
```

```
db.url=jdbc:mysql://localhost/howest-shop?serverTimezone=UTC
db.username=howest-shop-user
db.password=howest-shop-password
```

Fixing issue 1: taking config parameters out of Java code

```
public class MySqlConnection {

    private static final String KEY_DB_URL = "db.url";
    private static final String KEY_DB_USERNAME = "db.username";
    private static final String KEY_DB_PASSWORD = "db.password"; // NOSONAR

    private static String url;
    private static String username;
    private static String password;

    static {
        url = Config.getInstance().readSetting(KEY_DB_URL);
        username = Config.getInstance().readSetting(KEY_DB_USERNAME);
        password = Config.getInstance().readSetting(KEY_DB_PASSWORD);
    }

    private MySqlConnection() {
    }

    public static Connection getConnection() throws SQLException {
        return DriverManager.getConnection(url, username, password);
    }
```

- **Static** initialization block
- Executed **once** when class is **loaded**

Fixing issue 2: avoid storing credentials as clear text

- Username and especially password should be unreadable.
- We need some kind of two-way encryption.
  - We encrypt the credentials.
  - The encrypted credentials are stored in the .properties file.
  - When the application reads credentials from the .properties file, it should be able to decrypt them.
  - It can then use the decrypted credentials to connect to the database.

Fixing issue 2: avoid storing credentials as clear text

- We will be using the Spring Framework's Crypto library to apply two-way encryption (for storing in file) and decryption (when reading from file)
- https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/crypto/encrypt/package-summary.html
- Add dependency to **build.gradle.kts**:

```
implementation("org.springframework.security:spring-security-crypto:6.1.4")
```

Encrypting and decrypting text

- We need a TextEncryptor, which uses a secret (password) and salt

```
private static final String PASSWORD = "hello-from-howest";
private static final String SALT = "1AB9F37C2EDA";

private final TextEncryptor encryptor;

// …

private Crypto() {
    encryptor = Encryptors.text(PASSWORD, SALT);
}
```

Encrypting and decrypting text

- We can now use the encryptor to encrypt/decrypt:

```
public String encrypt(String in) {
    return encryptor.encrypt(in);
}

public String decrypt(String in) {
    return encryptor.decrypt(in);
}
```

Introducing the Crypto utility class

```
public class Crypto {
    private static final String PASSWORD = "hello-from-howest";
    private static final String SALT = "1AB9F37C2EDA";

    private static final Crypto instance = new Crypto();

    private final TextEncryptor encryptor;

    public static Crypto getInstance() {
        return instance;
    }

    private Crypto() {
        encryptor = Encryptors.text(PASSWORD, SALT);
    }

    public String encrypt(String in) {
        return encryptor.encrypt(in);
    }

    public String decrypt(String in) {
        return encryptor.decrypt(in);
    }
}
```

Introducing the Crypto utility class

```
public class Crypto {
   private static final String PASSWORD = "hello-from-howest";
   private static final String SALT = "1AB9F37C2EDA";

   private static final Crypto instance = new Crypto();

   private final TextEncryptor encryptor;

   public static Crypto getInstance() {
      return instance;
   }

   private Crypto() {
      encryptor = Encryptors.text(PASSWORD, SALT);
   }

   public String encrypt(String in) {
      return encryptor.encrypt(in);
   }

   public String decrypt(String in) {
      return encryptor.decrypt(in);
   }
}
```

Fixing issue 2: avoid storing credentials as clear text

- We can now encrypt database credentials in .properties file and decrypt at runtime:

```
db.url=jdbc:mysql://localhost/howest-shop?serverTimezone=UTC
db.username=ec8ef66311d61b773473d8677684ceca31dedf70
db.password=d4cc5ca938f144948ef66311d61b7734ec8ef663
```

```
public class MySqlConnect

// ...

    static {
        String usernameEncrypted = Config.getInstance().readSetting(KEY_DB_USERNAME);
        String passwordEncrypted = Config.getInstance().readSetting(KEY_DB_PASSWORD);

        Crypto crypto = Crypto.getInstance();

        username = crypto.decrypt(usernameEncrypted);
        password = crypto.decrypt(passwordEncrypted);

        url = Config.getInstance().readSetting(KEY_DB_URL);
    }
```

Problem with our solution…

- The Java class still contains the secret key in plain text…
- If a user decompiles the Java class or even opens it using a text editor, the secret key will be readable.
- Some additional solutions:
  - Store the secret key in a secure part of the operating system / server
  - **Ask the user for database username and password at runtime instead of storing in a configuration file**
  - Out of scope for this course, but make sure you know what the limitations of our solution are!