# Security Course

Version 2020

# Contents

# Security

## Introduction

JADE database systems contain valuable data and intellectual property, and can perform business-critical operations. As such, it is important to consider how that data and intellectual property is kept safe, and how those operations are kept running with high-availability and integrity.

When considering how best to implement and enforce security in JADE, you should start by asking the following three questions.

1. What specific assets do I need to protect?
2. What threats are possible that could compromise those assets?
3. What tools and best practices can I use to protect those assets from those threats?

For example, consider the JADE database system for a business that gives small business loans. What assets does this JADE system need to protect?

| Asset | Example Threat | How will I protect it? |
|---|---|---|
| Availability of core system | Tampering could bring down a Remote Access Program (RAP) or application server, preventing users from accessing the system | Control who has access to the system |
| Integrity of core system | An unauthorized user could approve loans he or she shouldn't be able to; for example, his or her own | Authenticate users to ensure they are who they say they are |
| Source code | A competitor could steal source code and copy the proprietary algorithm for determining whether to accept a loan | Encrypt the source code |
| User data privacy | A malicious actor could steal the customers' contact details and spam them | Encrypt data at rest and in transit |
| Data integrity | If I can make the system "forget" my loan, I could create an infinite money glitch | Control interfaces to the data to restrict what changes can be made |

## Security Threats

In this module, we will cover the common types of security threats, which can be remembered with the **STRIDE** acronym, as follows.

- **S**poofing
- **T**ampering
- **R**epudiation

- **I**nformation disclosure
- **D**enial of service
- **E**levation of privilege

## Spoofing

Spoofing is when a malicious actor impersonates another identity to gain access to which he or she is not authorized.

**Note** A *malicious actor*, also known as a threat actor, is a person or entity who is responsible for a negative security impact. This actor can be external or internal.

**Example**

A loan applicant logging in as a loan manager to approve his or her own loan.

## Tampering

Tampering is when a malicious actor modifies data in a way that causes harm.

**Example**

An insider connects to the JADE database server with an application server running the **JadeLogicalCertifier** application and deploys a malicious “fix” that corrupts data.

## Repudiation

Repudiation is when the authenticity or origin of a service or action cannot be proven.

**Example**

A developer logs onto the production environment and loads a schema, but it cannot be determined which developer it was because all developers share the same admin account.

## Information Disclosure

Information disclosure is when private or confidential information is leaked to anyone who does not have authority to view that information.

**Example**

A hacker gains access to the unencrypted data (**.dat**) files of the database and opens them in WinHex to get a list of emails and plain-text passwords.

## Denial of Service

Denial of service is when a malicious actor overloads a service with so many requests that it cannot keep up, preventing legitimate users from accessing the service.

**Example**

A botnet makes thousands of mal-formed web requests per second to the REST service, overloading the web servers and causing customers to be unable to access it.

## Elevation of Privilege

Elevation of privilege is when an application gains elevated access because of a bug or exploit.

**Example**

A JADE application compiles and runs a transient method that includes user input without proper sanitization. The user code does an injection attack to gain access to the Schema Inspector, which causes Information Disclosure.

**Note** In security, *sanitization* is the process of stripping or replacing special characters from user input, to avoid injection attacks.

## Discussion Questions

If you are going through this course self-paced, think about and write down your answers to the following questions. If you are going through the course instructor-led, your instructor will pose these questions for a class discussion.

Consider the following situations and determine which of the STRIDE security threats have occurred.

- While John is working on fixing a bug in a method called **filterLoansByCreditRating**, he notices a rather odd snippet of code.

```
// Ensure compliance
foreach loan in allLoans where loan.theBorrower.loans.size > 5 do
    // Only when all loans are last
    if loan.theBorrower.loans.last = loan then
        loan.theBorrower.loans.remove(loan); // needed for ISO requirement
    endif;
endforeach;
```

  This code doesn't make any sense to John, and it seems inappropriate in the context of filtering loans by credit rating. When he tried to find out who added the code and why, nobody in his team knew anything about it.

- After an unexpected reorganization is initiated at 2:38am, Jane is looking through the access logs to see who was online at that time. To her great surprise, the logs show that it was her! But she wasn't online at that time; she was fast asleep.

- Leigh is browsing Reddit late one Friday afternoon. Er, I mean, he's hard at work. Yup. But anyway, on Reddit he sees a link to a list of all his company's customers and their outstanding loan amounts.

## The Three 'A's of Access

The three '**A**'s is a set of key security concepts that govern the best practices for controlling access to sensitive data. They are:

1. **Authentication** – determining the identity of the user.
2. **Authorization** – determining whether the user is allowed to access the data.
3. **Accounting** (or Auditing) – tracking and logging what data a user accessed and any changes he or she made to the data.

You will often hear the terms *authentication* and *authorization* used interchangeably. There is a subtle difference between them, and we need to do both when deciding whether to allow a request to be executed on the server. The first thing we need to do is *authenticate* the security principle.

## Authentication

When we perform an authentication, we check the identity of the security principle; that is, we are verifying that they “are who they say they are”.

I want to come in, please.
May I see your ID, please?

Wilbur
Thanks, Wilbur!

## Authorization

After we know who it is who is attempting access, we then need to *authorize* them. This is when we check a set of rules to determine whether the (now-identified) security principle is allowed to access the resource.

Is Wilbur on the list?

## Accounting

After the user's identity has been authenticated and he or she has been authorized to access the data, we still have one more step: Accounting. This involves tracking and logging all data the user accessed and any actions the user takes. When accounting is done correctly, all changes to data should be able to be traced back to the source of that change, and there should be a record of what data each user has accessed.

This is important for a few reasons.

When planning an accounting process, it is helpful to first consider the goals of the accounting, as well as how to meet these goals, as shown in the example in the following table.

| Goal | To meet this goal, the… |
|---|---|
| Detect suspicious activity that may indicate an attack | Accounting process should not only monitor activity but also apply some rules to determine when that activity is suspicious and to generate a report. For example, a user who is authorized for a restricted set of activities attempts to perform every activity for which he or she is not authorized. This can indicate that the user is attempting to find a vulnerability and should automatically be reported to an administrator |
| Determine the impact of a security breach if it *does* occur | Process should keep a record of all data with which the user interacted (whether modified or only viewed). These records should not be stored in the database itself, as otherwise a malicious actor can delete them. |
| Provide evidence of criminal proceedings against a malicious actor | Detail and reliability of the records must be sufficient. |

# Applying the Three ‘A’s

This section covers applying the three ‘**A**’s to desktop applications and JADE REST services.

## Desktop Applications

You can use the **getAndValidateUser** and **isUserValid** methods of your schema’s **Global** class to authenticate and authorize the users of your JADE desktop applications.

Whenever a JADE application is started, the following authentication process occurs.

1. The **getAndValidateUser** method is called first.

   ```
   getAndValidateUser(usercode: String output; password: String
   output): Boolean;
   ```

   This method should typically be **clientExecution**, and is intended to obtain credentials from the user.

   The default implementation sets **usercode** to your workstation name suffixed with your operating system process ID, the **password** to null, and returns **true**. You can re-implement the method to replace this behavior with an implementation that obtains credentials from the user. Set the **usercode** and **password** output parameters based on this, then when you return from the method, these credentials are used to authenticate that user in the **isUserValid** method. You can also optionally perform some preliminary authentication in this method; for example, if you wanted to give the user multiple attempts before verifying on the server.

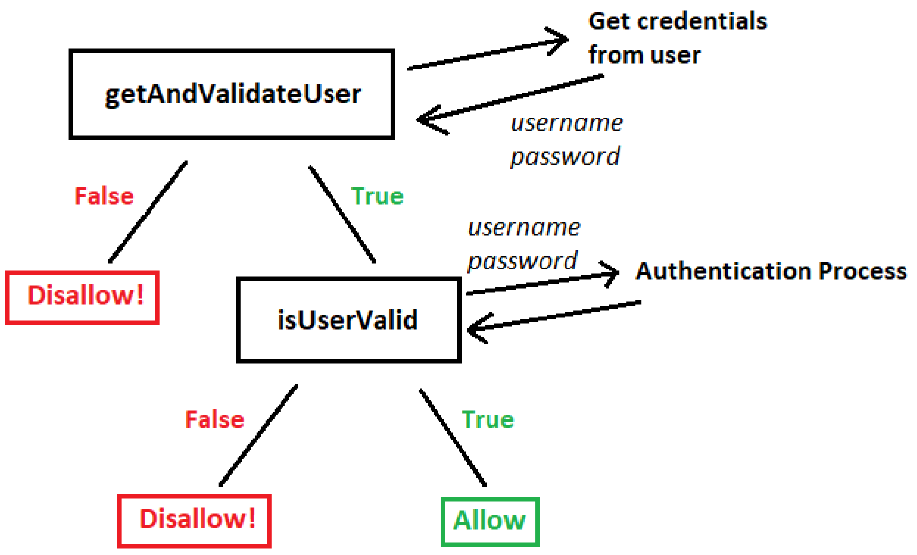
2. The **isUserValid** method is then called.

   ```
   isUserValid(usercode: String; password: String): Boolean;
   ```

   This method should typically be **serverExecution**, and is intended for authenticating the credentials.

   The default implementation simply returns **true**, but you can re-implement the method to replace this behavior with your authentication process. When you do, the **usercode** and **password** set in the **getAndValidateUser** method are passed as parameters.

3. If either method in step 1 or 2 returns **false**, the user is disallowed and the application will not start.



While you could implement the authentication process in JADE code, we want to avoid storing the user credentials in the JADE database. If user credentials are stored in the JADE database, it becomes your responsibility to ensure that they are encrypted with a strong and trustworthy encryption algorithm. It is easier and safer to delegate this responsibility to a trusted, re-usable solution from a reliable third-party vendor such as Microsoft Active Directory.

One way to do this is with the Application Programming Interface (API) provided by **CardSchema**, available free from the JADE Developer Center web site (https://www.jadeworld.com/jade-platform/developer-centre/download-jade).

**CardSchema** provides the **CnExternalMethods** class, which includes a **cnUserCheck** method. This method takes a username and password as input parameters, and an output parameter for the result. This method calls a Microsoft library that validates the username and password against the domain log ins, then sets the result to zero for successful validation or a non-zero error code for unsuccessful validation. Using this technique, we can authenticate the user without having to store user credentials in the JADE database.

**Note** Microsoft Windows login credentials will authenticate only the user (that is, verify who he or she is); the credentials will not be sufficient to authorize the user (that is, verify that the user is allowed to log on to the system).

## REST Web Service Security

You can secure your JADE REST services by requiring consumers to include JSON Web Tokens to authenticate their requests and associating required claims with your REST service methods to enforce authorization rules.

When dealing with the web, there are a lot of acronyms and terms with specific definitions. You can use the following as a reference for the terms we will use in this section.

| Term | Meaning |
|---|---|
| REST (Representational State Transfer) | An architectural style for web services where the server is stateless – the server forgets the client as soon as a request is fulfilled. |
| REST Service | An application that responds to REST requests over the web. |
| Security Principle | A person or program that requires access to a secured service. |
| Resource | The data or operation of the REST service that the security principle is attempting to access. |
| Consumer | The client of a REST service. Makes requests to the service and gets responses back. |
| JSON (JavaScript Object Notation) | A standard format for describing objects. |
| JWT (JSON Web Token) | A set of claims about a security principle plus a signature that proves that the token came from a trusted source. |

When a JADE REST service has not been secured, anyone who knows the correct Uniform Resource Locators (URLs) can access anything in your REST API. The traditional way of restricting access is by having a security principle log in, providing some secret or password to authenticate them, then authorizing them for the appropriate actions based on their identity.

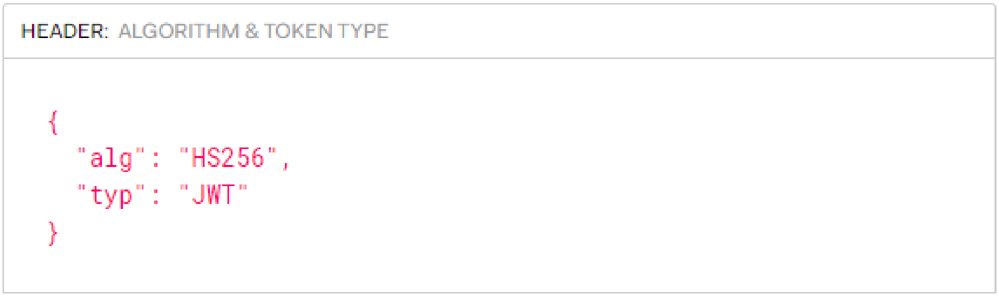
As REST services are stateless, there is an additional complication.

The server does not store any session information and therefore cannot determine whether the client is logged in. In lieu of this, we can provide the client with a security token after authentication. He or she can then include this token in future requests, removing the need to re-authenticate.

## What is a JSON Web Token?

A JSON Web Token is a string made up of the following segments.

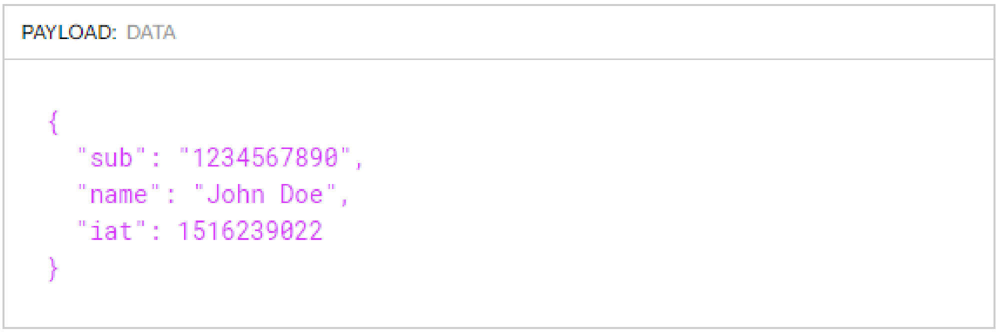
- Header, with meta-information about the token itself; for example:



HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

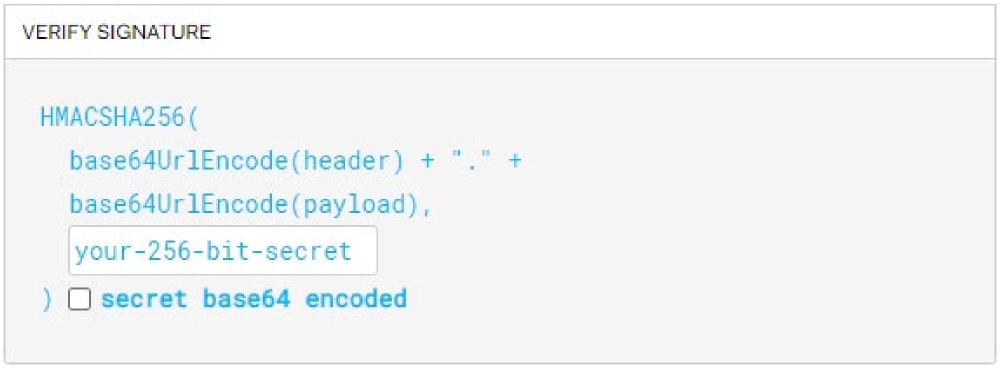
- Payload, with a set of claims about the identity of the bearer; for example:



PAYLOAD: DATA

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

- Signature, which proves that the token came from a trusted source; for example:



VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) ☐ secret base64 encoded
```

Each segment is then base64-encoded and delimited by a period, resulting in the following token.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZ
SI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36P
Ok6yJV_adQssw5c
```

In this token we have five claims. Two are in the header:

1. **"alg": "HS256"** – the token was signed using the HS256 signing algorithm.
2. **"typ": "jwt"** – the token is a JSON Web Token.

The header claims are meta-information; that is, claims about the token itself.

There are also three claims in the payload:

1. **"sub": "1234567890"** – describes a unique identifier for the subject of the token; that is, who the token is about.
2. **"name": "John Doe"** – the name of the subject of the token.
3. **"iat": "1516239022"** – the time at which the token was issued in the JSON **NumericDate** format, which is the number of seconds since midnight on the first of January 1970.

These payload claims are information about the subject of the token; that is, the security principle to be authenticated. There are many more claims you can put in the payload, such as the issuer of the token, the time at which the token should expire, and the audience of the token (by whom the token is intended to be validated). You can also make up your own claims to describe the security principle however you like, which can be useful when applying authorization rules.

Finally, the signature is generated by base64-encoding the header and payload, then encrypting the result with a secret, in this case using the HS256 algorithm and the secret **your-256-bit-secret**. When the server validates the signature, it can perform the same process on the header and payload, and if it was signed with a different secret or if anything in the header/payload has been modified, the signatures will not match and the token will be rejected.

## Symmetrical vs Asymmetrical Tokens

The symmetry of a token refers to the linked concepts of signing algorithm choice and whether it is the same entity that generates as validates the token.

In a symmetrically signed token, the token is generated and issued by the server itself and it will use a single secret that can encrypt and validate the token. This is the example we saw above. These tokens will use an HS encryption algorithm (HS256, HS384, HS512).

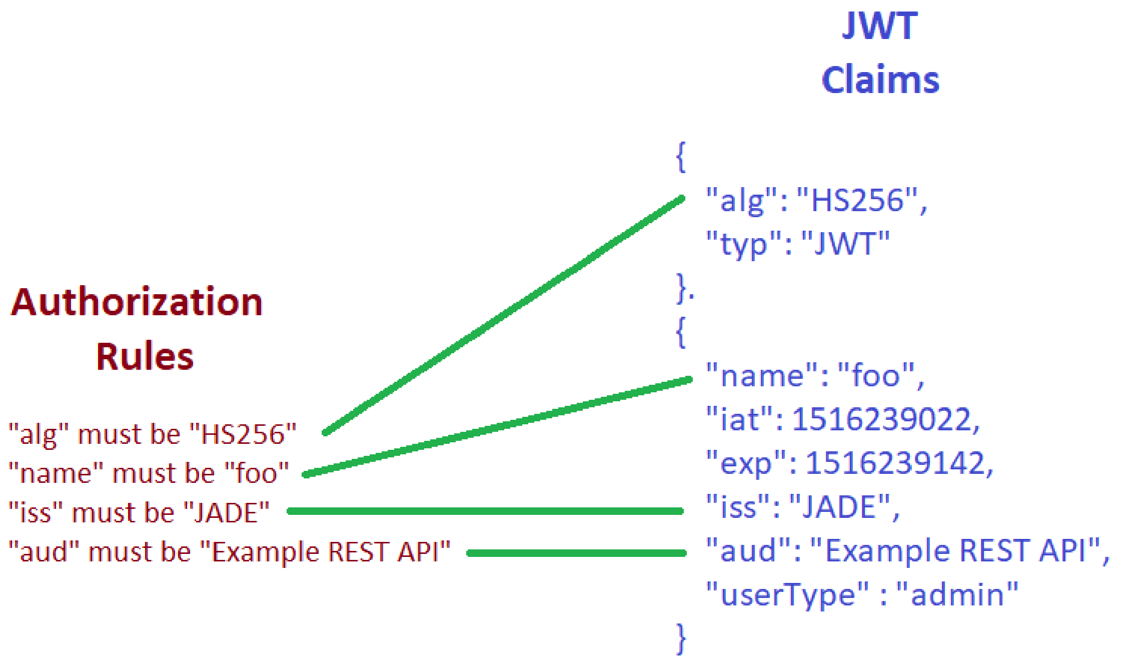
In an asymmetrically signed token, the token is generated by a trusted third-party and will be signed with a public/private key pair. This will allow the server to use the public key to verify the token while only the trusted third-party has the private key used to sign it. These tokens usually use an RS encryption algorithm (RS256, RS384, RS512) but there are other algorithms that are less common; for example, Elliptical-curve encryption.

## Generating a JSON Web Token from JADE

You can generate symmetrically signed JSON Web Tokens from JADE using the **JadeJsonWebToken** class.

## Enforcing Authorization Rules with JSON Web Tokens

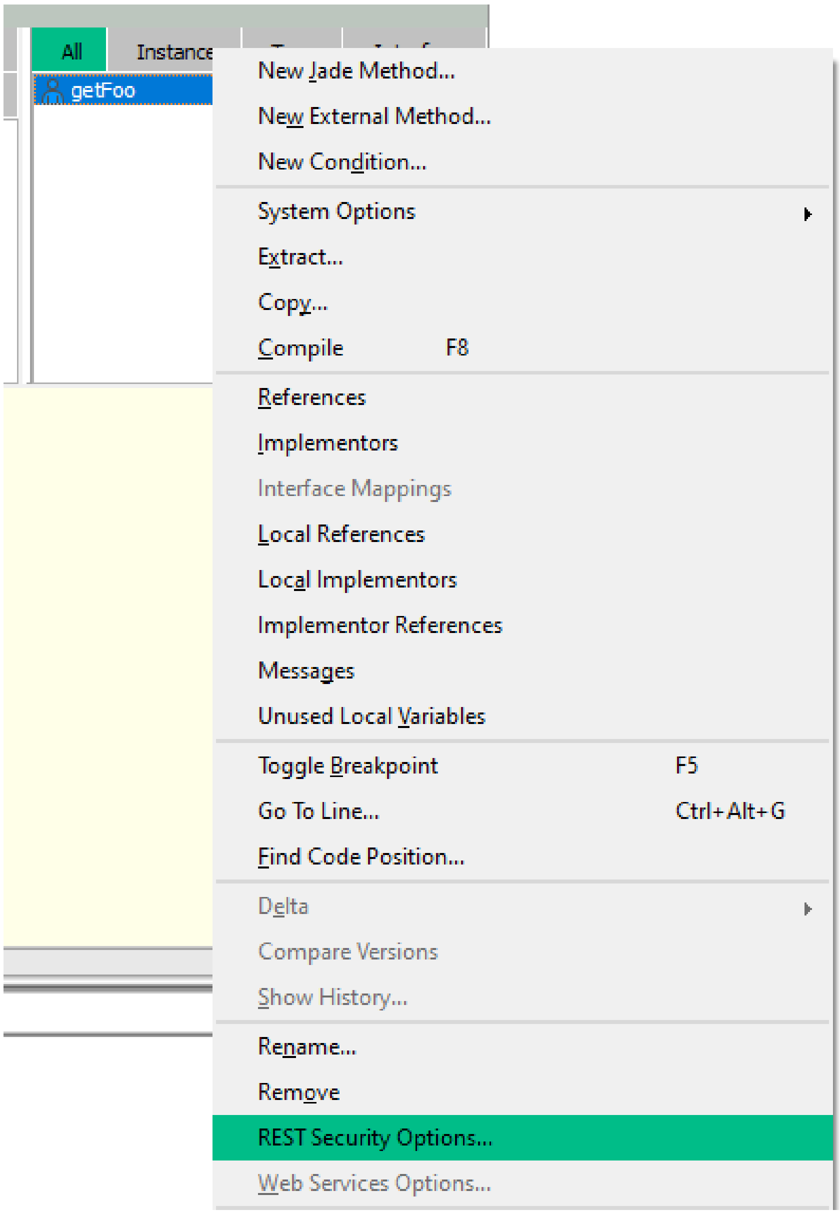
As mentioned previously, the purpose of a JSON Web Token is to allow the user to authenticate once only and obtain a token that can then be used to authorize multiple REST requests. This works by having a set of required claims associated with the REST resource and then comparing them with the claims included in a JSON Web Token that has been included in a REST request.



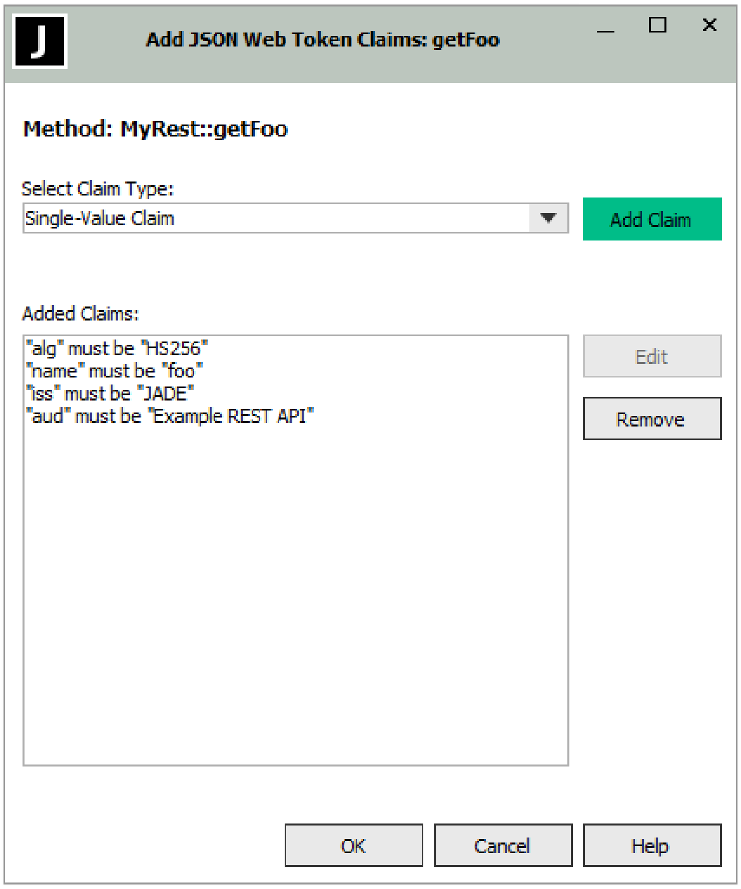
In this example, all of the authorization rules, also known as required claims, are present in the provided token. If the token’s signature is valid and the token hasn’t expired, the request will be allowed. Notice that the token also has a claim **"userType"**, which isn’t required by the rules. That’s allowed, as the token may have superfluous claims, as long as all of the required claims are present.

To associate a set of authorization rules with a REST resource method in JADE, we use the Add JSON Web Token Claims dialog.

- Select the **REST Security Options** command of the method's context menu, shown in the following image.



The Add JSON Web Token Claims dialog, shown in the following image, is then displayed.



Method: MyRest::getFoo

Select Claim Type: Single-Value Claim

Added Claims:

"alg" must be "HS256"
"name" must be "foo"
"iss" must be "JADE"
"aud" must be "Example REST API"

If any claims are associated with a method, all incoming requests must be authorized. If a request fails authorization, an HTTP response code 403 (*Forbidden*) will be returned.

To pass authorization, the following conditions must be met.

- A JSON Web Token must be present in the header of the request.
- The token's signature must be valid. How this occurs will depend on whether it is a symmetrically or asymmetrically signed token.
- All required claims must be present in the token.
- Default claims must be valid. For example, the expiry (**"exp"**) claim must be a time in the future.

## Mitigation of Potential Vulnerabilities

When using JSON Web Tokens, there are two main security vulnerabilities to know about, and both are mitigated for you by JADE.

- "alg": "none" vulnerability

  One of the options for the signing algorithm is **"none"**, where no signature is provided. The intent of this option is for testing purposes only, as any malicious actor could just send in such a token with whatever arbitrary claims he or she chooses. In JADE, the **"none"** algorithm is not supported and all tokens with **"alg": "none"** will be rejected. In fact, JADE uses a whitelist and only HS256, HS384, HS512, and RS256 algorithms are allowed. As such, you do not need to do anything to mitigate this vulnerability, as JADE does it for you.

- RSA vs HMAC vulnerability

  If a malicious actor sends a token he or she has created and set the **"alg"** claim to HS256 but actually pass in your auth provider's RS256 public key (which anyone can get, as it's public), some security libraries may treat the RS256 key as an HS256 key and erroneously accept it. By implementation, JADE is not vulnerable to this attack, so setting your **"alg"** as a required claim is entirely optional.

# Exception Handling and Deny by Default

When considering how to enforce authorization rules, it is not sufficient to consider only code paths that resolve as expected. It is also important to consider what will happen if an unexpected exception occurs.

The core principle for all authorization code is Deny by Default; that is, if the authorization code is unable to complete successfully, the default behavior should be to deny access rather than to allow it.

To illustrate this idea, we will consider some authorization methods, which will each call the following validation method.

```
validateUser(password : String) : Boolean;

begin
    return password = self.credentialStore.password;
end;
```

The important thing to notice about this method is that if the **credentialStore** reference is null, we will get a 1090 exception.

Consider this first method, in which we perform a simple authorization that should be allowed.

```
authenticationExample_ALLOWED();

vars
    validator : Validator;
    credential : Credential;

begin
    on Exception do handleException(exception); // returns "Resume Next"

    create credential transient;
    credential.password := "foobar";
    create validator transient;
    // We set the credential store.
    validator.credentialStore := credential;

    // The validateUser method will return true.
    if not validator.validateUser("foobar") then
        write "The user was forbidden.";
        return;
    endif;

    // So we will get this result.
    write "The user was authenticated";

epilog
    delete validator;
    delete credential;
end;
```

Everything is good so far, but let's see what happens if the passwords do not match.

**Note** As with all of the examples and exercises in this module, we will be looking at simple situations to illustrate one point at a time. You will notice that it is not doing anything when the user is authenticated; we are merely writing to the console log. In addition, we are faking the credentials by just using a transient object. The idea is to focus on the key concept of each example so that you can apply it to a wide variety of situations.

```
authenticationExample_FORBIDDEN();

vars
    validator : Validator;
    credential : Credential;

begin
    on Exception do handleException(exception); // returns "Resume Next"

    create credential transient;
    credential.password := "foobar";

    create validator transient;
    // We set the credential store.
    validator.credentialStore := credential;

    // The provided password is not the same as the stored credential.
    if not validator.validateUser("not foobar") then
        // So we are forbidden.
        write "The user was forbidden.";
        return;
    endif;

    // We don't get to here.
    write "The user was authenticated";

epilog
    delete validator;
    delete credential;
end;
```

As we would expect, the **validateUser** method will return **false** and the user is forbidden.

For simple cases, this implementation will therefore work *but* it fails the Deny by Default rule. Let's see what happens if we have an exception.

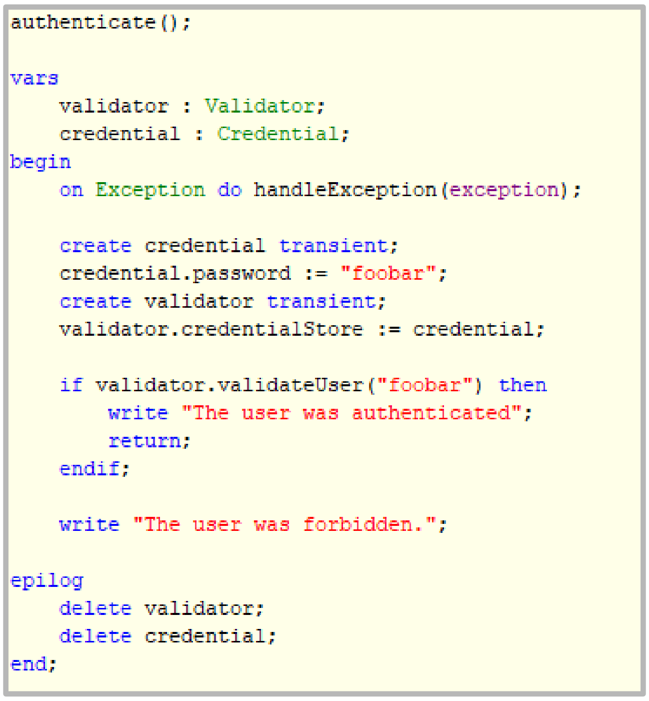
```
authenticationExample_EXCEPTION();

vars
    validator : Validator;

begin
    on Exception do handleException(exception); // returns "Resume Next"

    // This time we haven't set any credential to the validator so we will get a 1090.
    create validator transient;

    // This method will generate an exception and we will resume next, bypassing the whole if instruction.
    if not validator.validateUser("foobar") then
        write "The user was forbidden.";
        return;
    endif;


    // So we will end up here and erroneously authorize the user.
    write "The user was authenticated";

epilog
    delete validator;
end;
```

Here, when the **validateUser** method gets an exception, the exception handler's resume next bypasses the **if** instruction and we end up in the default code path. Since we did not Deny by Default, the user is erroneously authorized.

## Exercise 1 – Applying Deny by Default

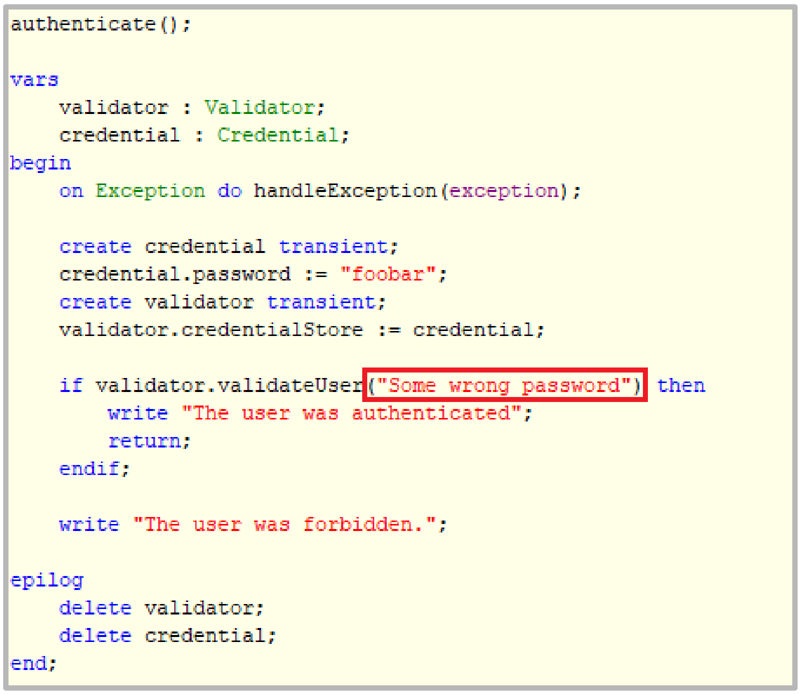
In this exercise, you will rewrite the **authenticationExample** method to deny access when an exception occurs.

1. Using the Schema Loader, load the **ExceptionSchema.scm** and **ExceptionSchema.ddx** files.
2. Add a new **JadeScript** method called **authenticate** and code it as follows.



```
authenticate();

vars
    validator : Validator;
    credential : Credential;
begin
    on Exception do handleException(exception);

    create credential transient;
    credential.password := "foobar";
    create validator transient;
    validator.credentialStore := credential;

    if validator.validateUser("foobar") then
        write "The user was authenticated";
        return;
    endif;

    write "The user was forbidden.";

epilog
    delete validator;
    delete credential;
end;
```

3. Execute the method. You should see that the user is authenticated.
4. Modify the **authenticate** method so that the password no longer matches.



```
authenticate();

vars
    validator : Validator;
    credential : Credential;
begin
    on Exception do handleException(exception);

    create credential transient;
    credential.password := "foobar";
    create validator transient;
    validator.credentialStore := credential;

    if validator.validateUser("Some wrong password") then
        write "The user was authenticated";
        return;
    endif;

    write "The user was forbidden.";

epilog
    delete validator;
    delete credential;
end;
```

5. Execute the method. You should see that the user is not authenticated. This is as expected, because the password does not match.
6. Now we will generate an exception to make sure we are not vulnerable to erroneous authorization when an exception occurs.
7. Navigate to the **Validator** class and modify the **validateUser** method.

```
validateUser(password : String) : Boolean;

begin
    write 1/0;
    return password = self.credentialStore.password;
end;
```

   The method will now generate an exception when it is run.

8. Execute the authenticate method (which should still have the invalid password). You should see that the user is still forbidden.

By having the default behavior being to deny access and then executing the code relating to a successful authorization only if the validation passes, we have made our code resilient to unauthorized access when things go wrong.

## Transient Methods and Code Injection

Transient methods are a powerful tool for dynamically creating JADE code at run time.

However, it is critical to ensure that the transient method is going to perform as you expect it to before execution, as it can be vulnerable to code injection attacks.

Consider the following method that creates and executes a transient method. It is a simple example, as the use of a transient method is superfluous but it serves to illustrate an important idea.

```
generateStatement(cust : Customer) : String;

constants
    SourceTemplate : String =
'createStatement() : String;
vars
    statement : String;
begin
    statement := "This is a statement for {CustomerName}" & CrLf & CrLf & "Your balance is: {CustomerBalance}";
    return statement;
end;
';

vars
    transMeth : JadeMethod;
    sourceCode : String;
    errCode, errPos, errLen : Integer;

begin
    sourceCode := SourceTemplate.replace__("{CustomerName}", cust.name, false)
                            .replace__("{CustomerBalance}", cust.balance.String, false);

    transMeth := process.createTransientMethod("createStatement", Customer, currentSchema,
                                    sourceCode, false, String,
                                    errCode, errPos, errLen);

    return process.executeTransientMethod(transMeth, cust).String;

epilog
    delete transMeth;
end;
```

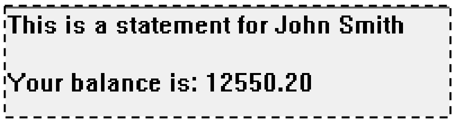
This method has a template for the source code of its transient method with two placeholders: **{CustomerName}** and **{CustomerBalance}**.

We are replacing these placeholders with values from a customer object, then creating a transient method with the source and executing it.

The **{CustomerBalance}** placeholder is set to the **balance** property of the customer. That's not an issue, because it is of type **Decimal** and decimals have a specific format that is not vulnerable to injection.

The **{CustomerName}**, however, is set to the **name** of the customer, which is a **String**. Strings can get up to all sorts of mischief.

If we call the method with a customer who has a name **John Smith**, the method will behave as expected and the transient method will generate a **String**.

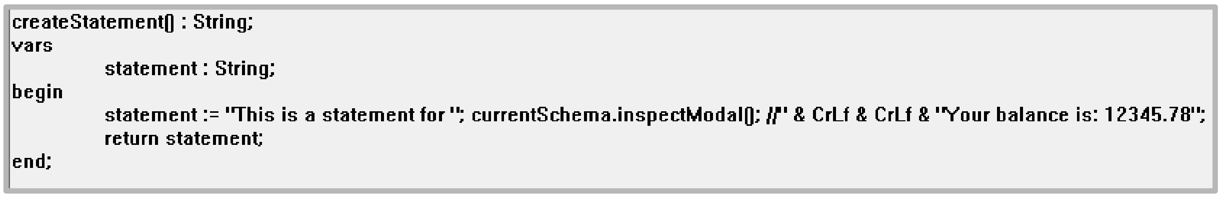


```
This is a statement for John Smith

Your balance is: 12550.20
```

So far so good. But what if the customer has the name:

```
"; currentSchema.inspectModal(); //
```

A bit of a funny name, but if we just let users type whatever they want for their name, nothing stops them from choosing such a name. Let's look at the source code for the transient method if we have that name.



```
createStatement() : String;
vars
        statement : String;
begin
        statement := "This is a statement for "; currentSchema.inspectModal(); //" & CrLf & CrLf & "Your balance is: 12345.78";
        return statement;
end;
```

The initial quote will escape the string that is supposed to be assigned to the **statement** variable, then the semicolon finishes that statement. It then executes the **currentSchema.inspectModal();** command, which will allow the user to see all information about all classes and objects in the entire schema. Alternatively, this could be any malicious code. Finally, it ends with **//** to comment out the rest of the line of code.

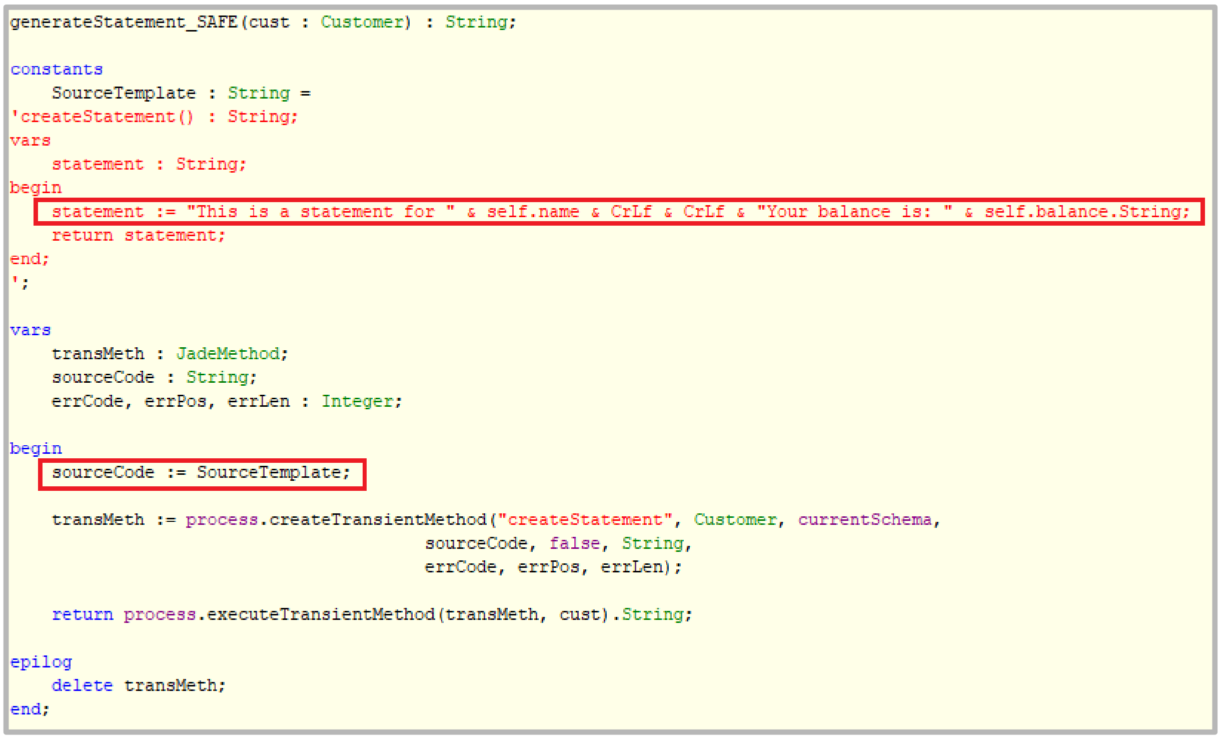
Using this method, the malicious actor can execute whatever code he or she wants to against the database.

## Mitigating Code Injection Attacks

There are two main ways to avoid code injection attacks to JADE transient methods. You can:

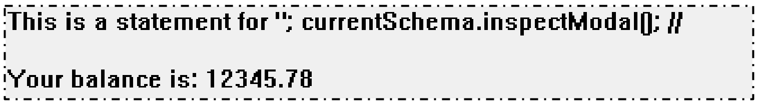
1. Keep user data away from transient methods.
2. Sanitize user data.

The first way is the safest and often the most elegant solution. For example, consider the following modified **generateStatement** method.



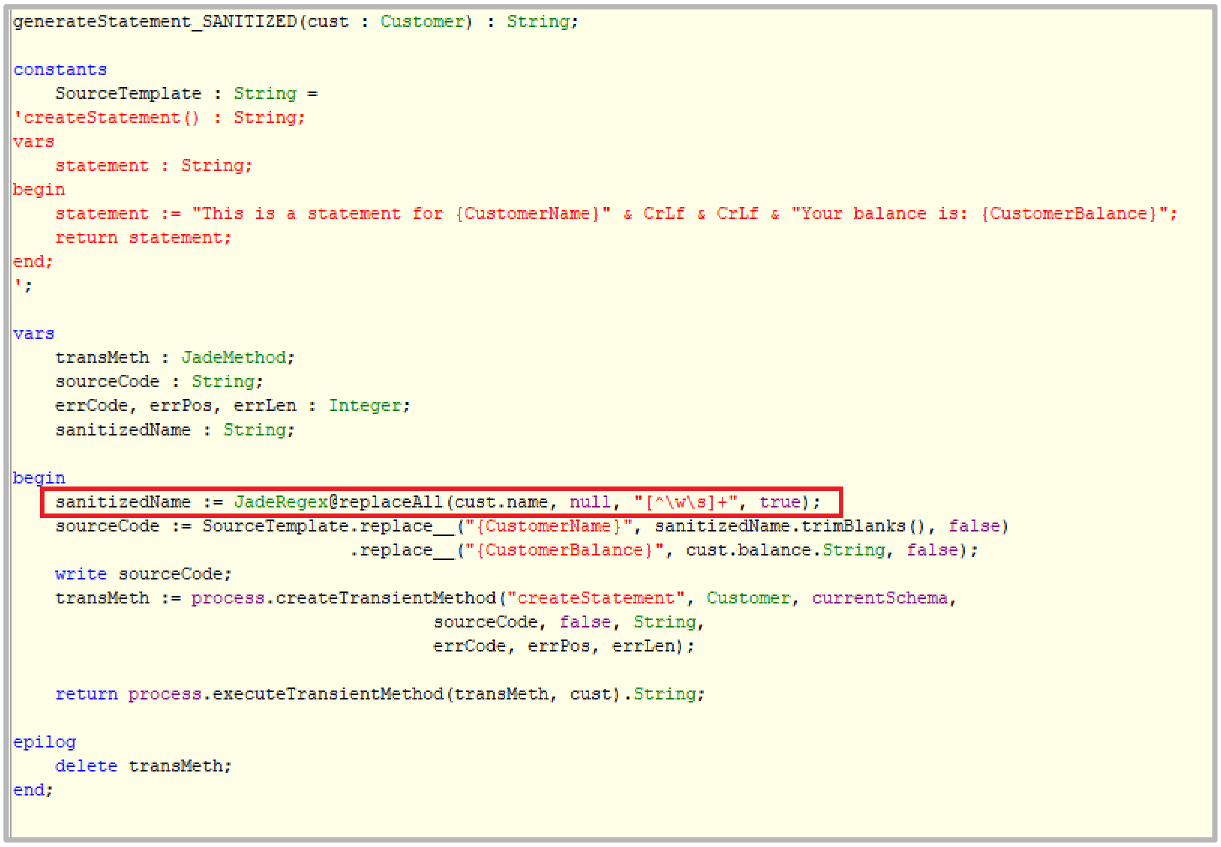
```
generateStatement_SAFE(cust : Customer) : String;

constants
    SourceTemplate : String =
'createStatement() : String;
vars
    statement : String;
begin
    statement := "This is a statement for " & self.name & CrLf & CrLf & "Your balance is: " & self.balance.String;
    return statement;
end;
';

vars
    transMeth : JadeMethod;
    sourceCode : String;
    errCode, errPos, errLen : Integer;

begin
    sourceCode := SourceTemplate;

    transMeth := process.createTransientMethod("createStatement", Customer, currentSchema,
                                    sourceCode, false, String,
                                    errCode, errPos, errLen);

    return process.executeTransientMethod(transMeth, cust).String;

epilog
    delete transMeth;
end;
```

This is again a fairly simple solution but illustrates the safety of not having any user data in the transient method source. Notice that we are now using the user data by referring to it with the **self** keyword. We can do this because the second parameter to **Process::executeTransientMethod** is the receiver for the method; in our case the customer object. The transient method will have access to the properties of that object, so we do not have to insert the values into the transient method source as string literals in the first place. With this change, we get the following output from our malicious input.



```
This is a statement for "; currentSchema.inspectModal(); //

Your balance is: 12345.78
```

The other way we can secure against this sort of attack is by sanitizing user data. To sanitize it, we strip any characters from user input that have a special meaning in JADE. For example, consider the following modified **generateStatement** method.



```
generateStatement_SANITIZED(cust : Customer) : String;

constants
    SourceTemplate : String =
'createStatement() : String;
vars
    statement : String;
begin
    statement := "This is a statement for {CustomerName}" & CrLf & CrLf & "Your balance is: {CustomerBalance}";
    return statement;
end;
';

vars
    transMeth : JadeMethod;
    sourceCode : String;
    errCode, errPos, errLen : Integer;
    sanitizedName : String;

begin
    sanitizedName := JadeRegex@replaceAll(cust.name, null, "[^\w\s]+", true);
    sourceCode := SourceTemplate.replace__("{CustomerName}", sanitizedName.trimBlanks(), false)
                                .replace__("{CustomerBalance}", cust.balance.String, false);
    write sourceCode;
    transMeth := process.createTransientMethod("createStatement", Customer, currentSchema,
                                        sourceCode, false, String,
                                        errCode, errPos, errLen);

    return process.executeTransientMethod(transMeth, cust).String;

epilog
    delete transMeth;
end;
```

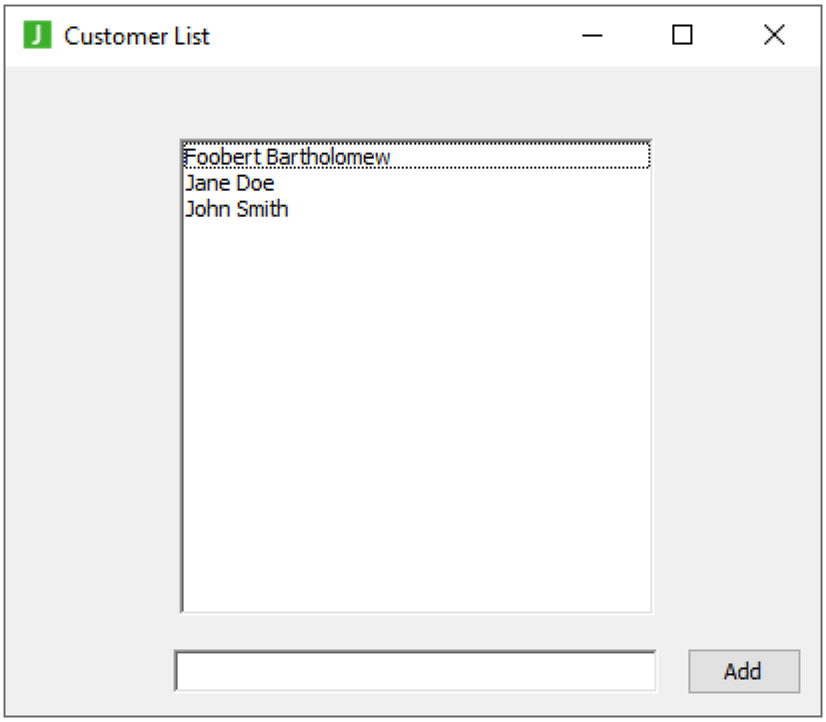
In this method, we have used a regular expression (Regex) to replace all characters other than alphanumeric (**\w**) and whitespace (**\s**) with nulls. This prevents any injected code from escaping the string and executing.

## Exercise 2 – Code Injection

In this exercise, you will use code injection to attack a poorly implemented system.

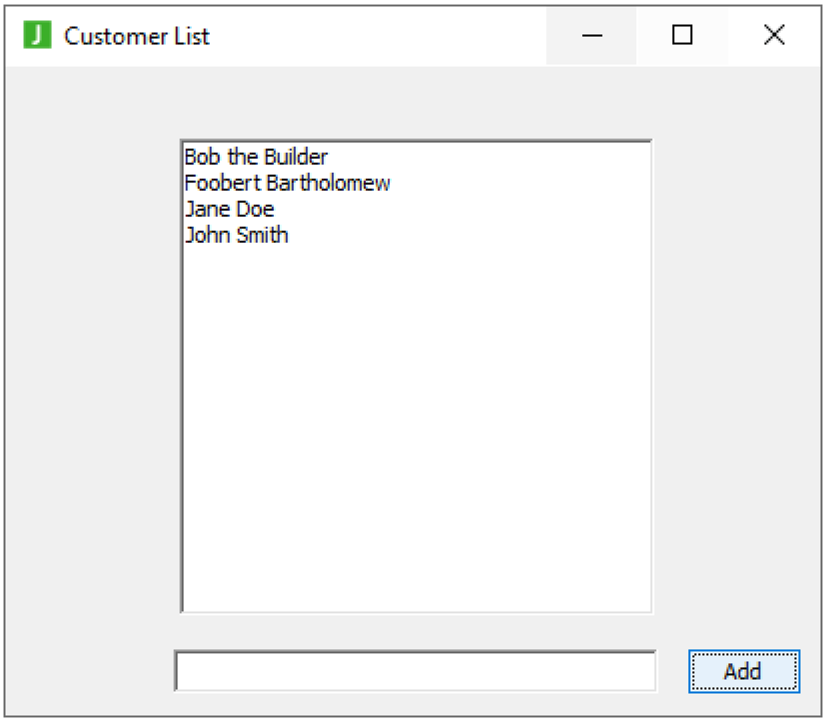
1. Close JADE if it is running, then restart it in multiuser mode; that is, start a database server and a fat client.
2. Use the Schema Loader to load the **TransientInjection.scm** and **TransientInjection.ddx** files.
3. From the **TransientInjection** schema, run the **TransientInjection** application.

The Customer List dialog, shown in the following image, is then displayed.



This simple application displays the names of some customers in a list box. In the text box at the bottom of the dialog, you can enter the name of a new customer, which is added to the list after you click the **Add** button.

4. Enter **Bob the Builder** into the text box and then click the **Add** button.



The list box in the Customer List dialog is populated in an unusual way. It uses a transient method to update itself whenever a new customer is added. I wonder what mischief can we get up to?

5. Try to do as much damage as possible to the database by entering malicious input for the customer's name.

   Use the following goals to get you started:

   - Try to bring down the fat client.
   - Can you access or delete data from another schema?