Search documentation Search

Appian

Accessing JNDI Data Sources in Plug-ins Best Practices

Appian 7.7

NOTE: This site does not include documentation on the latest release of Appian. Please upgrade to benefit from our newest features. For more information on the latest release of Appian, please see the Appian 7.8 documentation

The information on this page is provided by the Appian Center of Excellence

Toggle Na

This page is *not* supported by Appian Technical Support. For additional assistance, please contact your Appian Account Executive to engage with Appian Professional Services.

A data source in Java is the means of retrieving a connection to a database. A data source object will be registered with a naming service based on the Java Naming and Directory Interface (JNDI) API. Thus, to acquire a connection to the database, an Appian plug-in must obtain a JNDI data source connection.

This document shows the best practices for obtaining a JNDI data source connection from within a custom Appian plug-in. In all the examples a *valid* initial context must be acquired to retrieve a connection.

Before getting started, the JNDI name of a data source must already be known. The easiest way to retrieve this for an existing data source is through t Data Sources drop down on the Data Stores tab.

Best Practices

- Accessing the Appian Primary data source through JNDI it not supported, only accessing business data sources is allowed.
- It is recommended to use PreparedStatements where possible and avoid manually building SQL statements using strings which are susceptible to SQL injection attacks when user inputs are not properly sanitised.
- Resources must be properly managed to avoid starving the connection pool which can negatively affect system performance and cause an application outage.
- As connection pooling is used for JNDI connections, only request one when it is needed and release as soon as possible. Retrieving the connectio and keeping it open when it's not needed will tie up a connection unnecessarily and deprive another process of using it.
- Use the JNDI connection pool to retrieve a database connection, don't hardcode credentials or implement connection pooling within a plug-in.
- When executing multiple statements that could leave the database inconsistent if one fails transactions should be used.

Resource Management

Once a connection has been retrieved it must be closed when finished to ensure the pool is not starved of available connections. An application server c JDBC driver may attempt to close dangling connections automatically but that should not be relied upon.

Java 7 offers a convenient try-with-resources syntax to automatically close a resource when the code block completes normally or abruptly by exception. This syntax is cleaner than traditional code that uses a `finally` block to close connections.

```
try (Connection conn = ds.getConnection()) {
  try (PreparedStatement ps = conn.prepareStatement("SELECT name FROM customer")) {
    ResultSet rs = ps.executeQuery();
    while (rs.next()) {
      String name = rs.getString("name");
    }
  }
}
```

Notes

- In the above example the ResultSet is closed when the PreparedStatement is closed. If the PreparedStatement will be reused then the ResultSet should also use the try-with-resources syntax.

Function

The initial context can be created directly in Custom Functions.

Notes

- Unlike a Smart Service, the initial context is not injected and can be initiated directly.
- Do not confuse the Appian ServiceContext with the Java Naming Context used to acquire the JNDI data source connection.
- The read operation in this example is better handled by a Query function and is only shown for illustration purses.

- Functions should not be used to insert/update data due to potential data integrity issues. See writer function for more details. For inserts/updates use a writer function or Smart Service.

Example

```
@Function
public String getCustomerNameById(
  ServiceContext sc,
  @Parameter long id
) throws NamingException, SQLException {
  Context ctx = new InitialContext();
  DataSource ds = (DataSource) ctx.lookup("jdbc/AppianBusinessDS");
  String name = null;
  try (Connection conn = ds.getConnection()) {
    try (PreparedStatement ps = conn.prepareStatement("SELECT name FROM customer WHERE id = ?")) {
      ps.setLong(1, id);
      ResultSet rs = ps.executeQuery();
      if (rs.next()) {
        name = rs.getString("name");
      }
    }
  }
  return name;
}
```

Smart Service

The initial context must be injected into a constructor parameter for Custom Smart Services.

Notes

- Unlike Functions and Servlets, the initial context must not be initiated directly; it should only be acquired through injection.
- Do not confuse the Appian SmartServiceContext with the Java Naming Context used to acquire the JNDI data source connection.
- For reading data use a Function instead.
- This example of calling a stored procedure is intentionally simple and does not cover all use cases nor handle every edge/error case in a way that would be satisfactory for production use.

Example

```
@ConnectivityServices
public class UpdateCustomerName extends AppianSmartService {
  private SmartServiceContext smartServiceCtx;
  private Context ctx;
  private Long id;
  private String name;
  public UpdateCustomerName(SmartServiceContext smartServiceCtx, Context ctx) {
    this.smartServiceCtx = smartServiceCtx;
    this.ctx = ctx;
  }

  @Override
  public void run() throws SmartServiceException {
    try {
      updateCustomerName();
    } catch (NamingException | SQLException e) {
      throw new SmartServiceException.Builder(UpdateCustomerName.class, e).build();
    }
  }

  public void updateCustomerName() throws NamingException, SQLException {
    DataSource ds = (DataSource) ctx.lookup("jdbc/AppianBusinessDS");
    try (Connection conn = ds.getConnection()) {
      try (CallableStatement call = conn.prepareCall("{ call update_customer_name(?, ?) }")) {
        call.setLong(1, id);
        call.setString(2, name);
        call.execute();
      }
    }
  }

  @Input(required = Required.ALWAYS)
  @Name("CustomerId")
  public void setCustomerId(Long val) {
    this.id = val;
  }
```

```
  @Input(required = Required.ALWAYS)
  @Name("NewCustomerName")
  public void setNewCustomerName(String val) {
    this.name = val;
  }
}
```

Servlet

This example is a simple servlet that connects to a JNDI data source provided by the application server, executes an SQL query and iterates through th results.

Use Cases:

- Provide point access for a specific database report to an external system that is unable to integrate with the data source directly.
- Provide pull access to specific data in highly secure environments that limit direct access and where data cannot be pushed to an external data source.

Non Use Cases

- Using Appian to provide wholesale access to the underlying data source to an external entity.
- Integration with an unsupported database. This is better handled through custom functions and smart services.

Notes

- Unlike a Smart Service, the initial context is not injected and can be initiated directly.
- This example is intentionally simple and does not cover all use cases nor handle every edge/error case in a way that would be satisfactory for production use. Adapt the example as you see fit.

Example

```
public class DatabaseServlet extends HttpServlet {
  @Override
  protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
    PrintWriter pw = new PrintWriter(resp.getOutputStream());
    long id = Long.parseLong(req.getParameter("id"));
    String name = null;

    try {
      Context ctx = new InitialContext();
      DataSource ds = (DataSource) ctx.lookup("jdbc/AppianBusinessDS");
      try (Connection conn = ds.getConnection()) {
        try (PreparedStatement ps = conn.prepareStatement("SELECT name FROM customer WHERE id = ?")) {
          ps.setLong(1, id);
          ResultSet rs = ps.executeQuery();
          if (rs.next()) {
            name = rs.getString("name");
          }
        }
      }
    } catch (Exception e) {
      throw new ServletException("An error occurred while processing the GET request", e);
    }

    if (name == null) {
      pw.write("No customer found.");
    } else {
      pw.write("Customer Name: " + name);
    }
  }
}
```

User Guides by Role

Designer

Developer

Web Admin

Server Admin (On-Premise Only)

Tutorials

Records