Search documentation Search

Appian

Create Custom Data Types

Appian 7.7

Toggle U

Overview

Starting with Appian 6.2, custom data types and data stores are full-fledged Application Objects. You may create data types in the Data Type Design interface by importing XSDs.

Creating a Custom Data Type

Using the Data Type Designer

The Data Type Designer interface is accessible in the product by clicking System tab > Data Management folder > Data Management page > Create a Data Type toolbar link. The Data Type Designer allows you to visually create simple data types through the user interface. For all but the simplest use cases, however, you should click the Download XSD button instead of publishing the data type through the Type Designer. We don't recommend u just the Type Designer because it does not apply any of the JPA annotations necessary for proper management of data in a database.

Using XSDs

XSDs can be created by completing the following either of the following:

1. Downloading the XSD from the Type Designer.
2. Creating from scratch using an IDE such as Eclipse or a full-featured text editor such as EditPlus.

To determine which XSD constructs are valid for Appian data types, refer to the following Appian help page: Defining a Custom Data Type

The XSD file can then be imported using the Data Management interface.

Registering a Data Type from a Web Service

Custom Data Types can be created by parsing a web service WSDL. The Web Service node will automatically create the types necessary to consum the web service.

To create a data type from a web service, complete the following:

1. Drag a Call Web Service node onto the Process Modeler.
2. Open the Call Web Service node properties.
3. On the **Setup** tab, enter the WSDL URL and any necessary authentication information.
4. Click the **Get Services** button. Select the endpoint and the relevant web service method.
5. Review the Data tab. At this point, it should list the data types needed for the method's input and output parameters.
6. Save the Web Service node and the process model. This will register the data type with Appian. Publishing the process model is not necessary registering the data type.

To import other data types, select a different web service method in the node setup tab, save the new configuration, and then save the process mo You can repeat this for all custom data types to be registered.

Before importing the application to a new environment (such as moving from development to a staging or production environment), all data types n be registered beforehand by either importing an XSD or configuring and saving the Call Web Service node.

Updating Custom Data Type Definitions

Using the Data Type Designer

The Data Type Designer allows you to modify custom types in two ways:

- Adding new fields
- Changing field descriptions

If there is a structural change to the CDT, the Impact Analysis tool will be launched when you Save & Publish your changes.

To make any other change, you will need to follow the steps in the Using XSDs section.

Using XSDs

Custom data type definitions cannot be versioned explicitly by designers. To import a new version of a custom data type, the old version must be deleted from the Data Management interface and the new one imported. Importing can be done by uploading an XSD or via the Call Web Service node To import a new version using the Call Web Service node, follow the same instructions as when initially registering the data type through a web service.

After deleting the old type and importing the new type, use the Data Type Impact Analysis tool to update all Process Models and Data Stores that referenced the old type. A few things to note:

- The Impact Analysis tool will only find references to the old type inside Models and Data Stores that have been added to an Application. Models or Data Stores not in an Application will not be found by the Impact Analysis tool.
- Running processes are not updated - watch out for parent-child process model relationships that pass the process variables as references with CDTs because these might break if you update the data type in a sub-process model but not a running parent process.
- You may need to re-publish your data store

Updating Custom Data Type Definitions from WSDLs

If you are working with a web service that is still in development, it is possible that the types being referenced by the web service will need to change. When this happens, the above steps will not be sufficient because the product maintains a WSDL cache with the types.

You can clear the WSDL cache by restarting Jboss or by creating an XML file to disable the WSDL cache. The file should contain the following:

```
<cache-config>
   <cache-config key="webservices.wsdlCache"
         config="resources/appian/cache/jcs-wsdlCache-config.ccf"
         enabled="false"/>
   <cache-config key="webservices.wsInvokerCache"
         config="resources/appian/cache/jcs-wsInvokerCache-config.ccf"
         enabled="false"/>
</cache-config>
```

Name the file **wsdl-cache.xml** and place it in `suite.ear/web.war/WEB-INF/conf/cache`. Then follow the steps listed above.

Custom Data Type Best Practices

Most of these best practices revolve around adding JPA annotations. For more on JPA annotations, see Defining a Custom Data Type.

Assign Primary Key

In order to specify that a field is a primary key, you need to assign the @Id and @GeneratedValue annotations. If you do not add these annotations to field of your CDT, the product will create a primary key behind the scenes but it will not be exposed as a field of the CDT for you to use. Any update to CDT will result in a new record in the database being created.

Control Column Mapping and Constraints

Use the @Column annotation to specify a column name or add constraints. Use attributes such as the following:

- name
- length
- nullable (aka required)
- unique
- columnDefinition

See also: Documentation on Column Attributes

Keep in mind that constraints are enforced when the data is written to the database in the Write to Data Store node, not when the user enters the data or when the data is stored into a CDT Process Variable.

Relationship Mappings

TBD

Avoid Overwriting Shared Reference Data Types

When a parent CDT has a child CDT that contains reference or lookup data that may be modified by another process, use the @OneToOne(cascade=CascadeType.REFRESH) JPA annotation in the parent CDT definition to specify that the nested data is not to be overwritten when the parent entity is updated in the data store. Instead, the child CDT will be refreshed whenever the parent is written to the data store.

For example, a CDT of type Issue contains a nested CDT of type User. Many issues may reference the same user. The User data is also managed by a separate process which writes directly to the User data entity. The User information may therefore be changed outside of Issue causing the Issue data in a process to contain stale User data. The above annotation will prevent the stale data from overwriting the updated User data in the data store.

Take note that when this annotation is used, changes to the child CDT (e.g., User) must be saved to the data store by writing directly to the child CDT data entity (e.g., User) because any child data changes written to the parent data entity (e.g., Issue) will be disregarded

Referencing data in a separate table without a nested element

In order to reference select data from a nested CDT, the @SecondaryTable JPA annotation may be used. This will allow the data from the secondary table to be included in the parent CDT as a standard type and without the overhead of a whole separate nested structure. This can be useful for limiting the results of query rules because extraneous data from a child CDT can be excluded.

Using the above example of Issue and User, you can create a new IssueSummary CDT that includes data from the issue table and only the username from the userinfo table which is the table for the User CDT.

The IssueSummary CDT is annotated as follows:

```
@Table(name="issue")
@SecondaryTable(name="userinfo",
   pkJoinColumns=@PrimaryKeyJoinColumn(name="guid",
      referencedColumnName="username"))
```

In addition, the selected username field from the UserInfo table to be included in the IssueSummary CDT is annotated as follows:

```
@Column(name="username", nullable=false,
   insertable="false", updatable="false")
```

Prevent Race Conditions

Use the @Version attribute to define a field to prevent race conditions (e.g., two processes trying to update the same record in the database at the sam time, leading to overwritten data). This is referred to as Optimistic Locking. If a race condition is detected, the Write to Data Store node will pause by exception. A process administrator will need to intervene to resume the node once the race condition is manually resolved.

There are a few items to keep in mind when adding the @Version annotation.

1. This should only be used with Appian 6.6.1 or later (issue #42837)
2. The field must be a Number (Integer) type (i.e., xsd:int)
3. You should not update this value manually in Process - it will be updated automatically by the database whenever a Write to Data Store node updates the record. The latest value of the field will be returned in the Stored Value output of the Write to Data Store node.

Checklist

- □ Have all data types been defined as completely as possible for the process requirements before importing?
- □ Have XSD files for all data types not imported from web services been created?
- □ Have all XSDs been imported into the development site?
- □ Have all data types referenced by web services been imported using the Call Web Service Node?

© Appian Corporation 2002-2014. All Rights Reserved. • Privacy Policy • Disclaimer