Appian New Data Features

Serafeim Papastefanos
spapas@gmail.com

Features of Appian Versions

6.1	Data Stores	Query Rules	Write to Data Store	CDT Import/Export
6.2	CDT Designer	Multiple Analytics Engines		
6.5	Tempo	Appian Mobile		
6.6.x	Start Form Expression	Paging Grid	Increment constant	
6.7.x	Looping functions			
7	Some enchancements from Tempo			
7.2	SAIL	Constructing Data Type Values	Records	Tempo Reports
7.3	Enchancements to the above			

CDTs & Query Rules

- Instead of basic data types we can use Complex Data Types (structs)
  - CustomerInfoCDT
    - firstName
    - lastName
    - taxNumber...
- Each CDT can contain basic data types and other CDTs
- Grouping of information
- Easier storage & retrieval
  - Saving objects without ORM would require either a custom smart node or a subprocess
  -

- Query rules
  - Have to be defined
  - Can be used to retrieve lists of CDTs for usage in Processes
  - Can filter results (WHERE clauses in SQL)
  -

Creating CDTs

- Through CDT Designer from Appian GUI
  - Actually the CDT designer does not support JPA annotations and **cannot** be used alone!
- By writing an XSD by hand
  - Use the CDT designer to add the fields to the CDT
  - Download the XSD without publishing it to Appian
  - Edit it to add required annotations
  - Import it to Appian
- By importing types from web services
  - When you use the WS from Appian it will import all its types
  - These CDTs will be hidden at first but those that would be used can be unhidden
  - Recommended only on special cases - shouldn't be persisted
- By creating a java class in your custom plugin
  - It will contain both JAXB (for Appian) and JPA (for persistence) annotations
- A custom type will have:
  - A name
  - A namespace
  - I recommend unique type names by adding a prefix (PCP_Customer, KX_Customer)

Example CDT

```
<xsd:schema targetNamespace="http://appproxy.hcg.gr:8080/suite/types/"
  xmlns:types1="http://appproxy.hcg.gr:8080/suite/types/" <!-- Type namespece --> xmlns:xsd="
http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="Employee"> <!-- Type name-->
    <xsd:sequence>
      <xsd:element name="id" type="xsd:int"> <!-- ALWAYS add the primary key -->
        <xsd:annotation>
          <xsd:appinfo source="appian.jpa">
            @Id @GeneratedValue   <!-- The type should not used in Appian without a PK -->
          </xsd:appinfo>
        </xsd:annotation>
      </xsd:element>
      <xsd:element name="firstName" nillable="true" type="xsd:string"/>
      <xsd:element name="lastName" nillable="true" type="xsd:string"/>
      <xsd:element name="department" nillable="true" type="xsd:string"/>
      <xsd:element name="startDate" nillable="true" type="xsd:date"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Importing types from other CDTs

User.xsd

```
<xsd:schema targetNamespace="http://hcg.gr/appian/types/"
  xmlns:types1="http://hcg.gr/appian/types/" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="User"> … </xsd:complexType></xsd:schema>
```

Group.xsd

```
<xsd:schema targetNamespace="http://hcg.gr/appian/types/"
  xmlns:types1="http://hcg.gr/appian/types/" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:include schemaLocation="%7Bhttp%3A%2F%2Fhcg.gr%2Fappian%2Ftypes%2F%7DUser.xsd"/>
  <xsd:complexType name="Group">
    <xsd:sequence>
      ...
      <xsd:element maxOccurs="unbounded" minOccurs="0" name="members" type="types1:User"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

%7B = {, %3A%2F%2 =://, %7D = }

Example types

- Check XSDs of Sales reference application
- Map existing table to CDT
  - `@Table(name="us_states")`
  - `@Column(columnDefinition="CHAR(40) NOT NULL")`
- One to One

```
<xsd:element name="customerData" type="PCP_CustomerData">
      <xsd:annotation>
          <xsd:appinfo source="appian.jpa">
              @OneToOne(cascade=ALL, optional=false)
              @JoinColumn(name="customer_id", nullable=false, unique=true)
          </xsd:appinfo>
      </xsd:annotation>
  </xsd:element>
```

-

Example types 2

- **One to Many (order containing many order items)**

```
<xsd:element maxOccurs="unbounded" minOccurs="0" name="items" type="refapp1:SALES_OrderItem">
        <xsd:annotation>
          <xsd:appinfo source="appian.jpa">
                                @OneToMany(indexed = false)
                                @JoinColumn(name="orderid")
                           </xsd:appinfo>
       </xsd:annotation>
</xsd:element>
```

- **Many to One (a customer can have many orders)**

```
<xsd:element name="customer" nillable="true" type="refapp1:SALES_Customer">
    <xsd:annotation>
          <xsd:appinfo source="appian.jpa">
                                @ManyToOne(cascade = CascadeType.REFRESH)
                                @JoinColumn(name="customerid")
                            </xsd:appinfo>
    </xsd:annotation>
</xsd:element>
```

- **Many to Many (a customer can have many arrangement & an arrangmenet can have many beneficiaries/customers)**

Creating CDT with java class

- Appian types exported as XSDs *can be imported in eclipse*!
  - When other types are imported you have to include the XSD defining those types in the *same* directory and change the schema location!
  - JPA annotations are not created when importing however there is no public API to use JPA from plugins
    - So selects / updates must be performed using Appian query ruls & smart nodes
- Completely new appian types can also be created
  - Not recommended
  - Should be defined in the plugin with the datatype

```
<datatype key="ProjectDataType" name="Example Project Data Type">
        <class>com.acme.example.Project</class>
        <class>com.acme.example.Status</class>
</datatype>
```

Example of java CDT definition

```
@XmlRootElement(namespace="urn:my-namespace", name="status") // JAXB ANNOTATIONS
@XmlType(namespace="urn:my-namespace", name="status", propOrder={"id", "name"})
@Table(name="status")
public class Status implements Serializable {
    private Long id;    private String name;
    public Status(Long id, String name) {
        setId(id);        setName(name);
    }
    @Id    @XmlElement // JPA ANNOTATIONS
    public Long getId() {
        return id;
    }
    @Column(length=255, nullable=false, unique=true)    @XmlElement
    public String getName() {
        return name;
    }
}
```

CDT usage in plugins

```
// The SALESProduct class has been automaticall generated by importing it
@Function
@Type(name = "SALES_Product", namespace = "urn:com:appian:types:REFAPP") // Declare the return type
public SALESProduct[] TestQueryCDT(
      UserService us, // Inject the services that we are going to use
      @Parameter @Type(name = "SALES_Product", namespace = "urn:com:appian:types:REFAPP") SALESProduct
        salesProduct1,
      @Parameter @Type(name = "SALES_Product", namespace = "urn:com:appian:types:REFAPP") SALESProduct
        salesProduct2
) {
      ArrayList<SALESProduct> sps = new ArrayList<SALESProduct>();
      sps.add(salesProduct1);
      sps.add(salesProduct2);
      User user = us.getUser("serafeim");
      salesProduct1.setColor(user.getFirstName());
      salesProduct2.setDescription(user.getLastName());
      return sps.toArray(new SALESProduct[0]); // We can return Arrays of CDTs
}
```

* We tried returning a Dictionary Appian Type from a custom function (JsonToDict) but we were unable to instantiate a Dictionary (new Dictionary()) was not working. If we had this we could create another function that get a Json from a URL and then call all our JSPs from inside Appian without any more code

Instantiating CDTs

- We can instantiate CDTs using the type! function.
- Very important & useful in SAIL
- Example

```
type!SALES_Product(
  name:"product",
  description:"a product",
  color:"red",
  size:4,
  isActive:true,
  productCategory: type!SALES_ProductCategory(
    name:"category",
    description:"a category"
  )
)
```

Data stores

- CDTs can be added to datastores to be persisted in RDBMs
- A datastore is a collection of related CDTs
- Create one datastore per Process
  - All datastores could use the same datasource
- Tables can be either created automatically or through exported DDLs
- CDTs in datastores are called *Entities*
- Datastores have security!

Editing Types

- You can **only** add new fields to already published CDTs
  - Not needed fields would be always a part of a CDT
- When adding new fields the existing version would be “deleted” - renamed to CDT^2
- Old process instances would be updated to use the old/deleted version of the type (PersonData^2)
- Do an Impact analysis before changing CDTs:
  - https://forum.appian.com/suite/wiki/71/Data_Type_Impact_Analysis
- Be very careful with that
  - Bad behavior when passing CDTs by reference to sub process
    - Pass CDTs by value for long running sub processes
  - Bad behavior when updating CDT that is used in Smart Nodes
    - A new version of the plugin has to be created!

Usage within processes

- Write to Data Store Entity
  - Select the Entity
  - Add a new Node input with the correct type and CDT variable
  - Add a new node output (again with the correct type) to retrieve the persisted value
- Delete from Data Store
  - Create Process Variable with a Data Store Entity Type containg the type of the entity you want to delete
  - Keeps auditing information !

```
={
  {entity:pv!ENTITY_COMMIT, identifiers:pv!commitIdsToDelete},
  {entity:pv!ENTITY_SPRINT, identifiers:pv!sprintIdsToDelete}
}
```

- Add multiple entities to Data store (same as delete - data instead of identifiers)

Form usage - paging grid

- Should be used instead of grids
  - and dropdowns with many values
- Define the data set using the todatasubset function
  - Or a Query rule
  - Or by creating a custom function
  - https://forum.appian.com/suite/wiki/71/Paging_Grid_Component
- Define the CDT that will be used
  - Add columns for the fields of this CDT
- Users can do a (multiple) selection
  - The return value would be the primary key of the entity
  - Or the index of the array that was passed to todatasubset if no entities were used
- Supports filtering / sorting / paging

Advanced Querying

```
=queryrecord(
  recordType: cons!SP_CITY_RECORD,
  query: rule!APN_querySelection(
    fields: {"id", "name", "region.name", "region.district.name" },
    filter: rule!APN_logicalExpressionOR({
      rule!APN_queryFilter(
        field: "name",
        operator: "includes",
        value: ri!name
      ),
      rule!APN_queryFilter(
        field: "region.name", operator: "includes", value: ri!name
      ),
      rule!APN_queryFilter(
        field: "region.district.name", operator: "includes", value: ri!name
      )
    }),
    pagingInfo: ri!pagingInfo
  )
)
```

Use this for pagingGrid input !

Recommendations 1

- Each process should have **only one** CDT containing everything that needs to be persisted to the database (let's call it pdt)
  - TempeApplicationData
    - id (== IncidentID)
    - Protocol Number
    - TaskData (one2many)
    - TempeCustomerData (one2one)
    - TempeBuildingData (one2one)
    - TempoOtherData (one2one)
      - TempeOwner (one2many)
  - One to one relations could go to the main CDT to be more normalized
  - Write this to datastore after every user task
  - Use a query rule to retrieve the data of the process
- This prepares us for easier migration to **records (Appian 7.3)**

Recommendations 2

- CDTs still cannot be edited :(
  - Add an out of flow script task that “refreshes” the pdt from the database using a query rule
  - Whenever you want to do a dynamic intervention change the database values and run that flow
  - No need to edit the process model
- Try to avoid customizations (JSPs / javascript)
  - One exception could be the process info JSP page
- Everything should be done with Smart Nodes or custom functions
- Do not use Appian Grids - instead use paging grid for everything

Recommendations 3

- Start form should be avoided or contain only fields for search
  - Add a chained task *immediately* after the start node - no protocols would be generated and nothing will be persisted
  - If this task has not been submitted after 1 hour add an exception to end the process
- Customer Search should be performed through a paging grid and search fields (taxnumber, crs) contained in this form
  - User will fill the crs value and click on submit
  - The Custom Search will be performed through a Smart Node or custom function (better because it won't save data in the process)
  - The result will be shown in the paging grid - the user would select the customer and click submit
- Another form could follow to select the customer's arrangements
- All the above would replace the old, javascript heavy start forms

Recommendations 4

- New CDT versions *will* bite us.
  - Before doing anything in production do sanity checks in UAT
  - Pass CDTs by value in subprocesses with tasks
  - Plugins that take CDTs as a parameter must be updated (import types again) to work with new CDT version!