AAI in ONAP

- AAI delivered 13 Microservices in the Dublin release helm charts
- CRUD Microservices
  - Resources
  - Traversal
  - Graphadmin
  - Schema Service
- User Interface
  - Sparky (AAIUI)
  - Search Data Service
  - Data Router
  - Elasticsearch
- TOSCA Processing
  - Model Loader
  - Babel
- Other (future looking)
  - Gizmo
  - Champ
  - Spike
AAI in ONAP

• AAI has additional microservices that are not included in the Demo helm charts
  - Graphgraph
  - Cacher
  - Chameleon
  - Gallifrey

• During the Dublin release, the modeling subcommittee requested an initiative to reverse engineer the AAI data model

• To that end, the community has produced several different but complementary views of the AAI data model
• This presentation will cover the following:
  - AAI Graph Basics
  - The AAI Data Model Browser
  - GraphGraph
  - PlantUML
  - Papyrus
  - Custom Queries
  - DSL Queries
AAI Graph Basics

• A&AI uses **Janusgraph** for persistence which is a property graph model, where a graph is a set of vertices with edges between them.

• JanusGraph stores graphs in adjacency list format which means that a graph is stored as a collection of vertices with their adjacency list.
• The adjacency list of a vertex contains all of the vertex’s incident edges (and properties).
A vertex is the fundamental unit of the graph and represents an object. Vertex can have properties to describe the object.

**Vertex1**
- aai-node-type: Generic-VNF
- vnf-id: 8372929
- vnf-name: VNF1
- prov-status: Active
- model-id: 930183-09203-0039
- model-invariant-id: 384792
- nf-type: Type1
An edge is a connection between two vertices that expresses a relationship between them. An edge can have a multiplicity, direction, and properties.

```json
{
    "from": "vf-module",
    "to": "generic-vnf",
    "label": "org.onap.relationships.inventory.BelongsTo",
    "direction": "OUT",
    "multiplicity": "MANY2ONE",
    "contains-other-v": "!${direction}"
    "delete-other-v": "!${direction}"
    "prevent-delete": "NONE",
    "default": "true",
    "description": ""
}
```
Traversals discover and return information about edges, vertices, and their properties.

In graph databases, the relationship is a primary component of the data model and traversing from vertex to edge to vertex and beyond is the primary mechanism for querying the data within the model.
AAI Graph Basics

• Traversal Example: A custom query that starts at a generic-vnf, follows the edge to the service-instance, then gets all connected vnfs and their vf-modules.

• Also fetches the related tenant and vserver
AAI Data Model

- AAI uses a Graph Database based on JanusGraph / Titan
- AAI’s APIs are driven by its data model
- EclipseLink MOXy is the foundation of the AAI schema
- Types and their attributes are defined in OXM (Object to XML Mapping)
- Relationships – cousin and parent edges – are defined in a custom Edge Rules configuration (JSON)
- Nested objects and object reuse means that we have a huge number of endpoints
- Visualizing the data model is difficult
- Visualization from the swagger API is a messy eyechart
- Visualization without filtering out operational objects is also difficult
## Model Browser

- **Table-based view of the AAI data model**
- **Allows users to browse all installed versions of the AAI data model**
- **Provides navigation to browse nodes to which AAI has a**

---

**TO edges**

<table>
<thead>
<tr>
<th>Role</th>
<th>Type</th>
<th>Description</th>
<th>Key</th>
<th>Max</th>
<th>Req</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>String</td>
<td>Indicates the total bandwidth to be used for this service.</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>description</td>
<td>String</td>
<td>Short description for service instance.</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>environment-context</td>
<td>String</td>
<td>This field will store the environment context assigned to the service instance.</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>input-parameters</td>
<td>String</td>
<td>String capturing required parameters from ISO to pass to ClosedLoop.</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>model-invariant-id</td>
<td>String</td>
<td>The ASED model is for this resource or service model.</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>orchestration-status</td>
<td>String</td>
<td>Orchestration status of this service.</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>persona-model-version</td>
<td>String</td>
<td>The ASED model is for this resource or service model.</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>resource-version</td>
<td>String</td>
<td>Used for optimistic concurrency. Must be empty on create, valid on update and delete.</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>service-instance-id</td>
<td>String</td>
<td>Path to the controller object.</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>service-instance-locator-id</td>
<td>String</td>
<td>An identifier that customers assign to the location where this service is being used.</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>service-instance-name</td>
<td>String</td>
<td>This field will store a name assigned to the service-instance.</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>service-type</td>
<td>String</td>
<td>String capturing the service role.</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>updated-at</td>
<td>String</td>
<td>Last update of NetworkService.</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>vhn-portal-url</td>
<td>String</td>
<td>URL customers will use to access the VHN Portal.</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>widget-model-id</td>
<td>String</td>
<td>The ASED data dictionary widget model. This maps directly to the AAI widget.</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>widget-model-version</td>
<td>String</td>
<td>The ASED data dictionary version of the widget model. This maps directly to the AAI version of the widget.</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>workload-context</td>
<td>String</td>
<td>This field will store the workload context assigned to the service-instance.</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

**FROW edges**

<table>
<thead>
<tr>
<th>Role</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>collection</td>
<td>String</td>
<td>A collection of service instances.</td>
</tr>
<tr>
<td>service-instance</td>
<td>String</td>
<td>A service instance.</td>
</tr>
<tr>
<td>service-type</td>
<td>String</td>
<td>A service type.</td>
</tr>
<tr>
<td>service-instance-name</td>
<td>String</td>
<td>A service instance name.</td>
</tr>
<tr>
<td>service-instance-locator-id</td>
<td>String</td>
<td>A service instance locator id.</td>
</tr>
<tr>
<td>attr</td>
<td>String</td>
<td>A list of attributes.</td>
</tr>
<tr>
<td>description</td>
<td>String</td>
<td>A description of the service instance.</td>
</tr>
<tr>
<td>environment-context</td>
<td>String</td>
<td>An environment context.</td>
</tr>
<tr>
<td>input-parameters</td>
<td>String</td>
<td>String capturing required parameters from ISO to pass to ClosedLoop.</td>
</tr>
<tr>
<td>model-invariant-id</td>
<td>String</td>
<td>The ASED model is for this resource or service model.</td>
</tr>
<tr>
<td>orchestration-status</td>
<td>String</td>
<td>Orchestration status of this service.</td>
</tr>
<tr>
<td>persona-model-version</td>
<td>String</td>
<td>The ASED model is for this resource or service model.</td>
</tr>
<tr>
<td>resource-version</td>
<td>String</td>
<td>A version number for the resource.</td>
</tr>
</tbody>
</table>

---

Elapsed: 0.572093
• Pavel Paroulek has developed “Graphgraph” for the Frankfurt release of AAI

• Graphgraph examines the AAI schema and presents it in a User interface using React UI

• Graphgraph displays multiple dynamic views for understanding types and relationships in AAI
GraphGraph

- GraphGraph allows a very high view of the graph that shows the complexity of the schema and all the various ways the hierarchy can be arranged.
GraphGraph

• GraphGraph allows a user to zoom into a specific set of types and see how they are related

• In this example, you can visualize how the customer / service-subscription / service-instance objects are related
Keong Lim has developed a sequence of steps to parse the AAI data model and create Plant UML diagrams.

Below is an example schema which shows a hierarchical view of the OXM data, and shows the nesting / inheritance in the AAI schema.
• This provides an excellent visualization of the schema and the way objects are nested with BelongsTo relationships.

• Inventory is the top node and the root of the schema, and all the types are connected to it.

• In this example, you can see how the AAI URIs are created, a tenant’s URI is:

  /aai/vX/cloud-infrastructure/cloud-regions/cloud-region/{cloud-owner}/cloud-region-id/tenant/{tenant-id}
Plant UML – Edges View

• This chart shows how nodes are connected – as “cousin” edges
• These relationships are critical for traversals and are represented in the REST api in relationship-list objects
In this example, you can see that generic-vnf can be connected to a large number of other object types.
• Jacqueline Beaulac and Michela Bevilacqua have reverse engineered the AAI data model into papyrus UML.
• To the right is the diagram for the PNF object and the objects that it contains.
Looking closer you can see that it shows the relationships and hierarchy of the various types.

It also shows the attributes of each type, the type of each attribute and their cardinality.
Custom Queries

- Custom queries represent a sophisticated graph traversal mechanism which allows clients to develop and deploy stored queries to make processing complex services and ONAP deployments faster and more efficient.

- There is a proposal to add custom queries to graph graph so that they can be visualized – Volunteers?

- [https://wiki.onap.org/display/DW/Custom+Queries](https://wiki.onap.org/display/DW/Custom+Queries)
Custom Query

- Stored-queries with required/optional properties and query parameters
- PUT request on the query API
- Payload with start-node and query-name.
- Start-node -> URI or an array of URIs or Nodes API
- Format = id / simple / pathed / resource / resource_and_url / graphson / count / console
- Optional Query parameters: nodesOnly, subgraph
Custom Query Input

```json
{
  "start": ["{namespace}/{resource}"],
  "query": "query/{query-name}"
}

{
  "start": ["{namespace}/{resource}"],
}

{
  "start": ["{namespace}/{resource}"],
  "query": "query/{query-name}?prop1=value1&prop2=value2"
}
```
Custom Query Example – Closed Loop

**Closed-loop**: Start with `vserver` and retrieve data required for closed-loop action (replaces a closed-loop-named-query)

```json
{
    "start": "/cloud-infrastructure/cloud-regions/cloud-region/{cloud-owner}/{cloud-region-id}/tenants/tenant/{tenant-id}/vservers/vserver/{vserver-id},
    "query": "query/closed-loop"
}
```

**Traversal to use**: `vserver`

- `vserver > generic-vnf`
  - `generic-vnf > model-ver`
    - `model-ver > model`
    - `generic-vnf > service-instance`
      - `service-instance > model-ver`
        - `model-ver > model`
      - `service-instance > generic-vnf`
    - `generic-vnf > vf-module`
      - `vf-module > model-ver`
        - `model-ver > model`
  - `vserver > tenant`
    - `tenant > cloud-region`
Custom Query - Related-to

**Related-to:** Start with any starting node and the query returns all related nodes of a requested node-type with optional edge-type

```json
{
    "start": "...",
    "query": "query/related-to?startingNodeType={node-type}&relatedToNodeType={node-type}"
}
```

OR

```json
{
    "start": "...",
    "query": "query/related-to?startingNodeType={node-type}&relatedToNodeType={node-type}&edgeType={edge-type}"
}
```

**Traversal to use:** `{starting-node-type} > {related-node-type}`

**Stored-queries.json**

```json
{
    "related-to": {
        "query": {
            "required-properties": ["startingNodeType", "relatedToNodeType"],
            "optional-properties": ["edgeType"]
        },
        "stored-query": "builder.createEdgeTraversal(edgeType, startingNodeType, relatedToNodeType).store('x').cap('x').unfold().dedup()"
    }
}
```
Custom Query Syntax continued..

Get generic-vnfs from pserver: Start with pserver hostname or fqdn and retrieve the generic-vnfs related to it. This query also supports pre-filtering the vnf results using optional parameters.

```
{
    "start":["nodes/pservers?hostname=<hostname>"],
    "query":"query/genericVnfs-fromPserver?vnfType={}"}

OR

{
    "start":["nodes/pservers?fqdn=<fqdn> "],
    "query":"query/genericVnfs-fromPserver?vnf-type={}&nf-function={}"}
```

Traversal to use: pserver > generic-vnf
              pserver > vserver > generic-vnf
Custom Query – genericVnfs-fromPserver

```json
{
    "genericVnfs-fromPserver": {
        "query": {
            "optional-properties": ["vnfType", "nfFunction", "nfRole", "nfNamingCode"]
        },

        "stored-query": "builder.union(
            builder.newInstance().createEdgeTraversal(EdgeType.COUSIN, 'pserver', 'generic-vnf'),
            builder.newInstance().createEdgeTraversal(EdgeType.COUSIN, 'pserver', 'vserver').
            createEdgeTraversal('vserver', 'generic-vnf').
            getVerticesByProperty('vnf-type', vnfType)
            getVerticesByProperty('nf-function', nfFunction)
            getVerticesByProperty('nf-role', nfRole)
            getVerticesByProperty('nf-naming-code', nfNamingCode)
            .store('x').cap('x').unfold().dedup()"
        )
    }
}
```
Formats

- **Id**: Resource link includes the vertex Id

  "results": [
    {
      "resource-type": "generic-vnf",
      "resource-link": "/aai/v$/resources/id/2388112"
    }
  ]

- **Pathed**: Resource link includes the uri of the vertex

  "results": [
    {
      "resource-type": "generic-vnf",
      "resource-link": "/aai/v$/network/generic-vnfs/generic-vnf/lab20105v"
    }
  ]

- **Resource**: Same format as resources API response payload with depth=1
- **Resource_and_url**: resource format plus the pathed url
- **Graphson**: Results in graphson format
- **Count**: Provides count of objects in the query
- **simple**: node-type, graph vertex id, pathed url, object properties, and directly related objects in the graph are all returned.

```json

"results": [
{
    "id": "739696712",
    "node-type": "generic-vnf",
    "url": "https://<host>:8443/aai/v$/network/generic-vnfs/
generic-vnf/85f60b5e-6eff-49c8-9a79-550ee9eb4806",
    "properties": {
        "vnf-type": "WX",
        "vnf-name": "ONAPHLPA0703UJWX01"
    },
    "related-to": [
    {
        "id": "739700808",
        "node-type": "license",
        "url": "https://onap:8443/aai/v10/
        network/generic-vnfs/generic-vnf/
        85f60b5e-6eff-49c8-9a79-550ee9eb4806/
        licenses/license/ONAP-M/
        VONAP-81"
    }
    ]
}
```
DSL Queries (Bring Your Own Query)

- DSL (Bring Your Own Query) is a new feature being delivered in Frankfort that will allow users to specify ad-hoc queries that will make AAI more flexible and robust than previous versions.
- Like with custom queries, there is a proposal to add DSL queries to the graph graph views.
BYOQ: A simple DSL to describe a graph traversal query
provides the abstraction to translate to underlying graph language (gremlin, cypher)
BYOQ language was built using Antlr4 (AAIDSL.g4).
2-step translation which gets translated first to our A&AI internal DSL and next to underlying graph query language using Groovy
BYOQ UI (part of Sparky UI) is under development, where users can build queries and retrieve the results
Formats and query parameters similar to custom queries
PUT request on the DSL API
https://<host>:8446/aai/v$/dsl?format={format}&<optional-query-parameters>
• Input Payload

```json
{
    "dsl" : " customer('global-customer-id','8310000058863-16102016-aai1539')
       > service-subscription > service-instance
       > connector* > virtual-data-center* > generic-vnf* 
}
```

Starting with a customer object, traverse through service-subscriptions and service-instances, and traverse any edges to connector objects. Store the connectors and any connected virtual data centers and their generic vnfs, and return the set.
DSL Syntax

Node Query (Filter based on properties or negation)
node* (‘key’,value’) (‘key2’,’value2’). . .  
* → store(x) or ‘Select *’
node* !(‘key’,value’) (‘key2’,’value2’). . .  ‘!’ → neq or negation

Example:
cloud-region* (‘cloud-owner’,’onap-rs1’)(’cloud-region-id’,’onap3’)

Traversal with Node Query
node1(‘key1’,value1’) > node2 > node3* (‘key3’,value3’)

Example:
cloud-region (‘cloud-owner’,’onap’)(’cloud-region-id’,’ONAP25’) > availability-zone*

Where clause (Filter based on traversal )
node1* ( > node2(‘key’,’value’))
node1* (‘key1’,’value1’) ( > node2!(‘key2’,’value’))

Example
pnf* (> complex(‘physical-location-id’,’ONAPDATA’))
Traversal Query with union

- node1('key','value')+ > [ node2> node3*, node3* ]
- For a Topology view, when you want to view a sub-set of nodes from NODE1

Example:

```
cloud-region('cloud-owner','onap-rs1')('cloud-region-id','onap3') >
[ complex*('state','NJ'),
  l3-network*('network-type','tst-EXAMPLE_BASIC_NETWORK')
]```

Best Practices for CQ and BYOQ

- Verify that the start element or the start node has (this is the preferred order)
  - URI
  - Keys
  - Unique indexed properties
  - Indexed properties
- Run cq2gremlin API in traversal (https://<host>:8446/aai/cq2gremlin) and get the gremlin equivalent. Run a profile on the gremlin equivalent to make sure the gremlin query is optimal
- Graph or Schema Service could help with the shortest path to define the optimal traversal for the custom query
- Do not start a query with a union. Union queries do not use indexes
Best Practices for CQ and BYOQ

• Use subgraph=* sparingly. It tries to get the related edge for every vertex returned by your custom query. It would get very expensive if the vertex has too many related edges (like cloud-region).
• Use groupCount sparingly or in UI queries. Not for use by clients in application.
• groupCount by a property is equivalent of a full table scan. The only index that it uses is aai-node-type which is a low cardinality index.
• Frequently used DSL queries should be encouraged to be converted to custom-queries since the start-node in a custom query has a URI which is uniquely indexed in the graph.
• Run a count on the query and verify if the traversal can be reversed to get the optimal query.
Rest Layer

- Rest Consumer classes are the entry point for all Rest endpoints in the microservice
- Validates the request and completes the preprocessing required for the request
- Applies query parameters (depth, nodesOnly) to the request
- Formats the response payload – calls core library to transform response payloads

- QueryConsumer: REST layer for custom query API (/query)
- DSLConsumer: Rest layer for DSL queries(/dsl)
- RecentsAPIConsumer: Rest layer for recents API (/recents)
- SearchProvider: Rest layer for nodes and generic query(/search)
- ModelAndNamedQueryRestProvider: Rest layer for model query and named query (/model)

- Separate consumer class for each endpoint
- Register in JerseyConfiguration.java(org.glassfish.jersey.server.ResourceConfiguration)
Gremlin QueryBuilder

- Provides abstraction from the A&AI schema and from the underlying graph query language
- Converts the A&AI internal DSL to gremlin

Key Methods

- `getVerticesByProperty(key, value) → has('key', 'value')`
- `getVerticesByProperty(key, MissingOptionalParameter value)`
- `getVerticesByProperty(key) → has('key')`
- `createEdgeTraversal(EdgeType, nodeType1, nodeType2)`
  - Find if edgerule exists between the nodeTypes. If yes, finds the direction for the traversal `.out().has(‘aai-node-type’, ‘nodeType2’)` or `.in().has(‘aai-node-type’, ‘nodeType2’)`
- `createEdgeTraversalWithLabels(EdgeType, nodeType1, nodeType2, List<> labels)`
  - `.out('edgelabel').has('aai-node-type', 'nodeType2')` or `.in('edgelabel').has('aai-node-type', 'nodeType2')`

Implemented constructs

- Union, where, has, hasNot, select, or, store, cap, unfold, dedup, emit, repeat, both, tree, by, path
Query Builder Internals

• { "start" : ["cloud-infrastructure/cloud-regions/cloud-region/{cloud-owner}/{cloud-region-id}"]},
  "query" : "query/availabilityZoneAndComplex-fromCloudRegion"

• builder.union(builder.newInstance().createEdgeTraversal(EdgeType.TREE, 'cloud-region', 'availability-zone').store('x'),builder.newInstance().createEdgeTraversal(EdgeType.COUSIN, 'cloud-region', 'complex').store('x')).cap('x').unfold().dedup()

• g.V(vertices).has('aai-node-type', 'cloud-region').has('cloud-owner', 'onap').has('cloud-region-id', 'ONAP25').union(__.in('org.onap.relationships.inventory.BelongsTo').has('aai-node-type', 'availability-zone').store('x'),__.out('org.onap.relationships.inventory.LocatedIn').has('aai-node-type', 'complex').store('x')).cap('x').unfold().dedup()

• GraphStep([],vertex), HasStep([aai-node-type.eq(cloud-region)]), StoreStep(x), HasStep([aai-node-type.eq(service-subscription)]), StoreStep(x), StoreStep(x), VertexStep(OUT,[usesL3Network],vertex), HasStep([aai-node-type.eq(I3-network)]), StoreStep(x), VertexStep(IN,[uses],vertex), HasStep([aai-node-type.eq(cloud-region)]), StoreStep(x), SideEffectCapStep([x]), UnfoldStep]

• CQ2Gremlin : API that will help you verify the gremlin for your custom query