MarkLogic Server

Entity Services Developer's Guide

MarkLogic 9
May, 2017

Last Revised: 9.0-9, April 2019

Table of Contents

Entity Services Developer's Guide

1.0 Introduction to Entity Services

Business analysts often describe processes in terms of logical business entities, such as Customers and Orders, and the relationships between them. MarkLogic Entity Services is a set of tools and interfaces that make it easier to create applications that manipulate these business entities, even when your raw data has a different structure.

You can use Entity Services to model your business entities and generate code and configuration artifacts that facilitate creating, querying, and exporting entity instances.

This section contains the following topics:

1.1 Terms and Definitions

The material in this guide assumes the reader is familiar with the following terms and definitions:

Term	Definition
model descriptor	A definition of a set of entity types, their properties, and relationships. You use a descriptor to create a model and model-based application code and configuration artifacts. For more details, see “Creating and Managing Models” on page 57.
model	A model includes entity type definitions, entity property definitions, relationships between entity types, and facts about the model (as semantic triples). A model descriptor contributes the entity type and entity property definitions, and relationships between entitites. MarkLogic generates a default set of facts from the descriptor, and you can add additional facts to the model. For details, see “Creating and Managing Models” on page 57.
entity	An abstraction of a logical business object that can be stored and manipulated by applications. For example, a sales model might include entities such as a customer, order, or inventory item.
entity type	A definition of the characteristics of an entity instance, including its properties and relationships to other entities.
entity instance	A concrete instantiation of an entity type, as represented by a populated data structure representing an individual entity, or a document containing such a data structure.
entity property	A concrete characteristic of an entity type. For example, a customer entity type might have properties such as a name, address, and customer id. Entity properties whose type is an entity type express an entity relationship.
entity relationship	A logical relationship between entity types. For example, an order entity type might include relationships with a customer and inventory item entities. In Entity Services, an entity relationship is expressed as an entity property whose type is an entity type (rather than scalar or array type). For details, see “Defining Entity Relationships” on page 80.
envelope document	By Entity Services convention, a document that encapsulates an entity instance, metadata, and, optionally, the raw source from which the entity was generated. For details, see “Managing Entity Instances” on page 149.

Term	Definition
local reference	In a model descriptor, a reference to an entity type that can be fully resolved within that descriptor. For example, if a model defines Race and Runner entity types, and a Race entity type has a property that is an array of references to Runners, then those references are local references. For details, see “Defining Entity Relationships” on page 80.
external reference	In a model descriptor, a reference to an entity type that is not defined within the same descriptor. For details, see “Defining Entity Relationships” on page 80.
TDE template	A Template Driven Extraction (TDE) template. Use Entity Services to generate a template that enables querying your entity instance data as rows or semantic triples. For details, see “Generating a TDE Template” on page 122 and “Search Basics for Instance Data” on page 171.
harmonization	The process of transforming data from disparate sources into a common, model-based representation.
data hub	An application that takes in raw data from disparate sources and transforms the data into canonical business entities that can be used by applications without regard to differences in the original source.

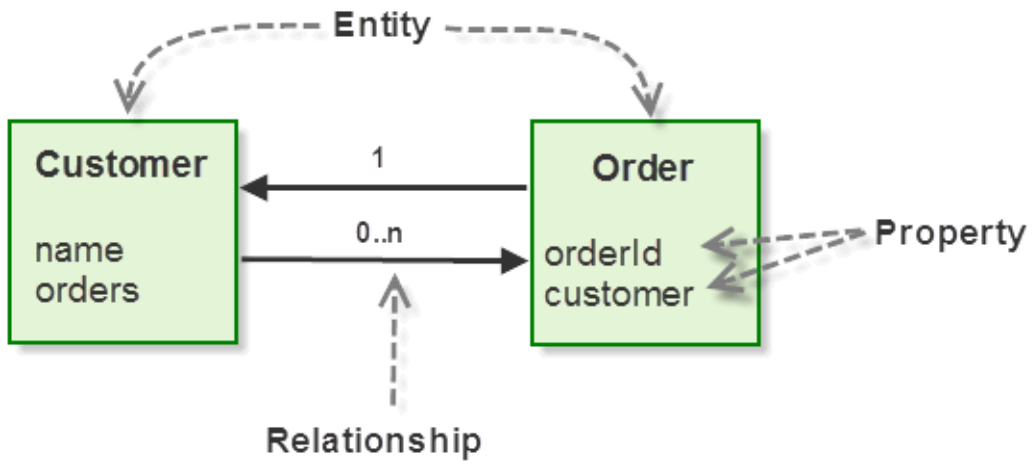
1.2 Why Use Entity Modeling?

Enterprise applications must often work with data from multiple sources. The data shares common conceptual objects, such as “customer” or “order”, but representation details can differ significantly. The “meaning” of the data is spread across schemas, application code, ETL code, and the minds of developers, DBAs, and data stewards.

Working directly with this heterogeneous data imposes cognitive load on developers and adds complexity to applications. A model-based view of your data eliminates these problems because it surfaces a consistent view of the “real world” objects and relationships in your data, independent of the raw representation.

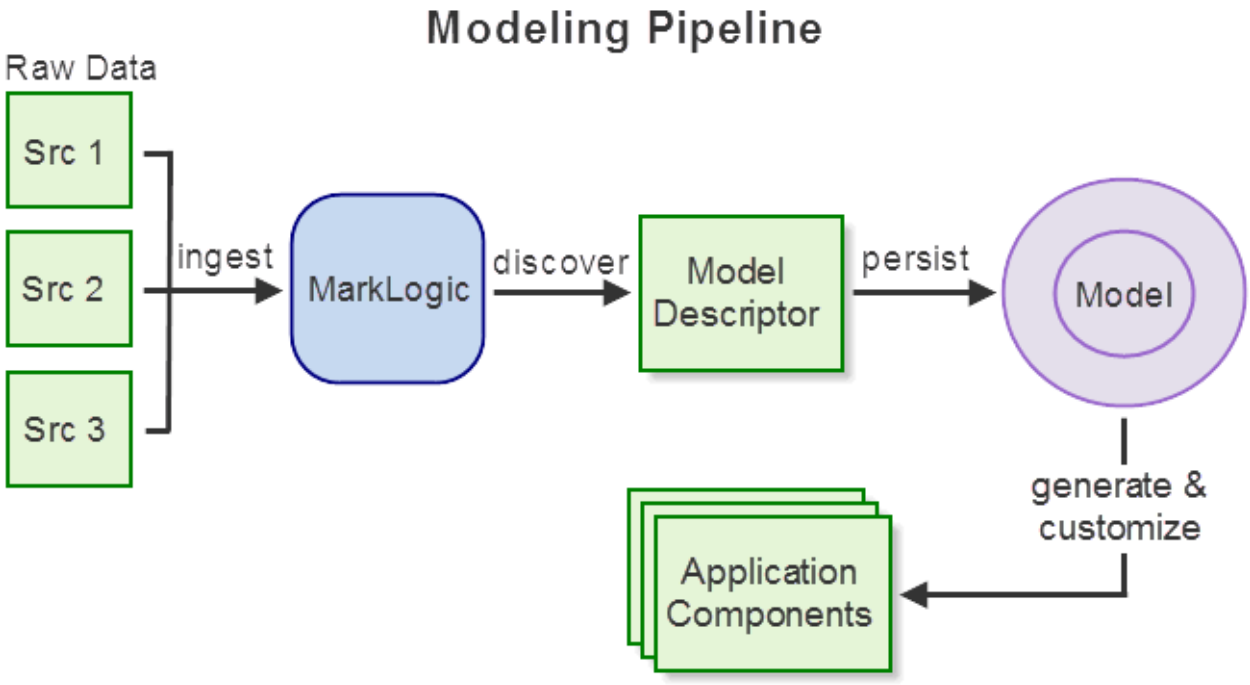
A model defines logical entity types, their properties, and the relationships between entities. For example, say your model includes logical “customer” and “order” entities. A customer entity includes a “name” property. An order entity includes an “order number” property. There are relationships between customer and order entities: A customer is associated with each order, and a customer has a list of a orders.

You might capture this information in a modeling diagram such as the following:



Entity modeling fits well with MarkLogic. You can ingest your heterogeneous raw data and immediately get value out of it, using MarkLogic's application development, search, and indexing features. These same features enable you to explore your data for purposes of data discovery. As you explore your data, you uncover entities and relationships that can be modeled.

Using the Entity Services API, you can capture your modeled entity types, properties, and relationships in a model descriptor, and then use the descriptor to create a model. Given a model, you can use Entity Services to generate a variety of artifacts on which to build your model-based application. The diagram below outlines this process. For more details, see “Entity Services Overview” on page 11.



You can build up a model iteratively. You do not need to finalize your model to begin getting value from the model or your data. The model can grow and change as your data does, without negatively impacting downstream data consumers: Model based code can easily accomodate a new data source or a new data discovery, such as the need to expose a new entity type.

Modeling also enables you to expose different views of your data. For example, if you are modeling patient data, you might have one model that exposes a billing view of the data and another model that exposes a “quality of care” view of the data. Both models can sit on top of the same raw data set and need not be defined simultaneously.

1.3 Entity Services Overview

Entity Services is an API and a set of conventions you can use to quickly stand up an application based on entity modeling.

The Entity Services API provides the following services to facilitate application development based on entity modeling:

- Modeling Vocabulary: The modeling vocabulary supported by Entity Services provides a structured way to describe entities, their properties, and relationships between entities.

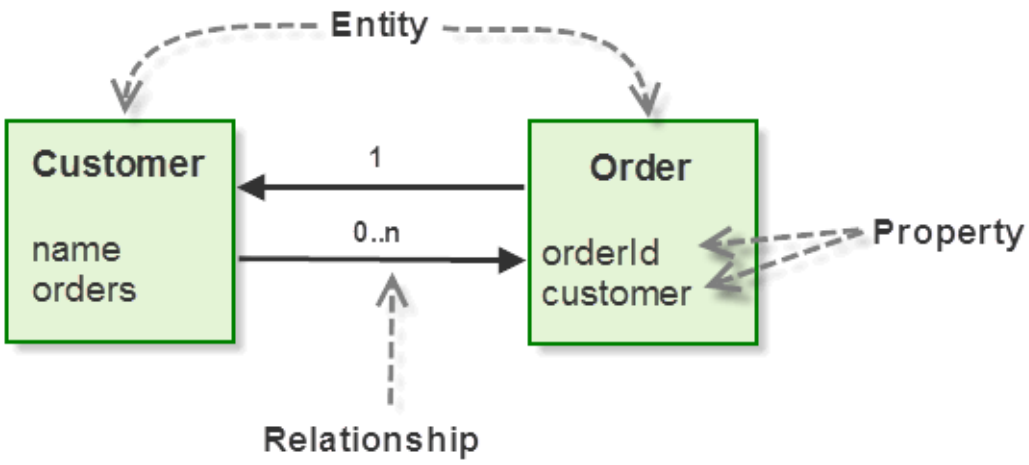
- Persistence Convention: The entity persistence pattern promoted by Entity Services defines a convention for representing harmonized entities, metadata, and raw data as documents. Your applications can centralize on a single pattern for storing and manipulating entities.
- Application Scaffolding: You can use Entity Services to generate code and configuration artifacts from an entity model. This provides a well-defined framework on which to base an application.

Entity Services promotes a convention for implementing model-based applications, but it does not force this convention on you. For example, you can use the API to generate code for encapsulating entity instances, metadata, and raw source in an envelope document with a recommended structure. However, you are free to modify or replace this structure.

1.3.1 Modeling Vocabulary

Entity Services supports a modeling "vocabulary" in the form of a model descriptor. The descriptor syntax is based on Swagger and JSON schema. A model descriptor not only identifies entity types, their properties, and relationships, but also captures information such as data types and metadata.

For example, recall the entity diagram from "Why Use Entity Modeling?" on page 9:



This diagram captures entity types and relationships, but does not include data type and other details required by a developer. Entity Services uses a model descriptor to capture detailed entity type definition and metadata in one place. This enables data stewards and developers to share a common view of the model.

The model descriptor is the basis for creating a model, generating code templates, and generating schemas and configuration artifacts. An Entity Services model descriptor can be expressed in either XML or JSON.

A JSON descriptor for the above diagram might look like the following. Metadata about the model is captured in the "info" section, while the entity types, their properties, and relationships are captured in the "definitions" section.

```
{ "info": {
    "title": 'OrderTracker',
    "version": '1.0.0',
    "baseUri": 'http://acme.com/sales/',
    "description": 'A model of customer order tracking'
  },
  "definitions": {
    "Customer": {
      "properties": {
        "name": { "datatype": 'string' },
        "orders": {
          "datatype": "array",
          "items": { "ref": "#/definitions/Order"}
    } } },
    "Order": {
      "properties": {
        "orderId": { "datatype": "string" },
        "customer": { "ref": "#/definitions/Customer"}
    } },
} }
```

You can express additional requirements, such as which properties are required and which properties should be indexed for efficient search.

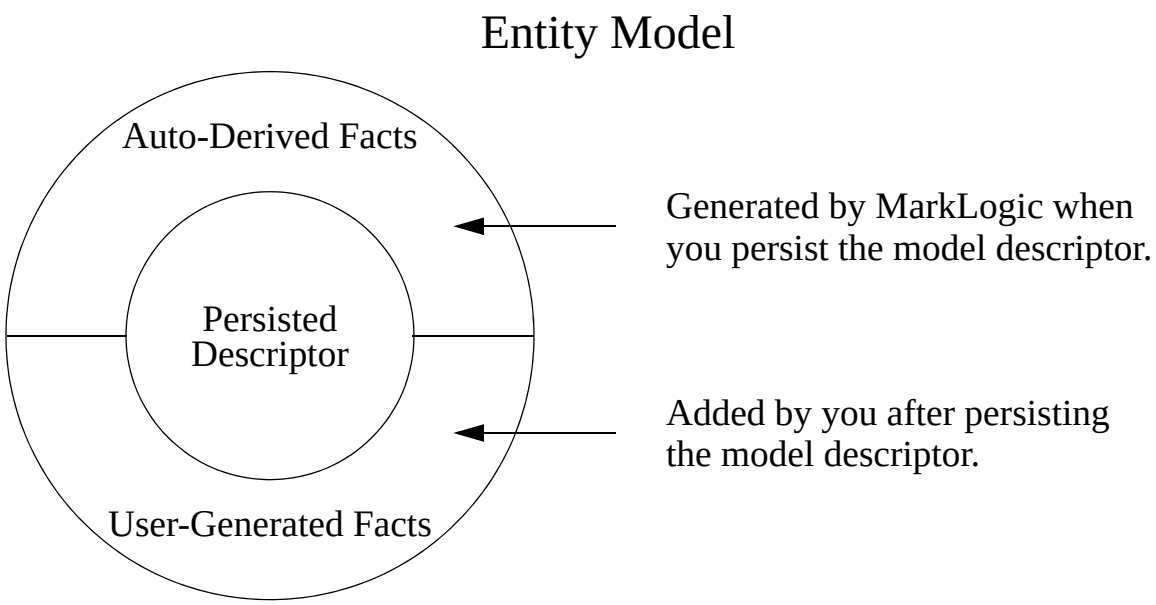
For more details, see "Creating and Managing Models" on page 57.

1.3.2 Persistence Convention

When you follow the Entity Services paradigm, you persist two kinds of modeling related artifacts in the database: The model and entity instance envelope documents.

When you persist a model descriptor in MarkLogic as a document in the special Entity Services collection, MarkLogic generates a model from the descriptor. This model is a graph of semantic triples representing "facts" about the model. The initial set of facts are those that can be derived from the model descriptor. You can then extend the model to include your own facts, in the form of additional triples. For more details, see "Creating and Managing Models" on page 57.

The following diagram depicts the key parts of an entity model:



By convention, an entity instance is persisted in MarkLogic as part of an envelope document that encapsulates the instance, instance metadata, and the raw source data from which the instance is derived. You manage envelope documents like any other document in MarkLogic. You can use Entity Services to generate some configuration and other artifacts that facilitate searching instance data stored in recommended envelope layout. For more details, see “Managing Entity Instances” on page 149.

1.3.3 Application Scaffolding

Once you create a model, you can use it with Entity Services to generate code, schemas, and configuration artifacts to help you create a model-based application. The generated code and artifacts are designed to be customized and extended to meet the needs of your application. Entity Services does not enforce any particular data layout or code pattern.

You can generate the following code modules using Entity Services. The input in all cases is a model descriptor. You are expected to customize the generated code to meet the needs of your application.

- Instance Converter Module: A code template for converting raw source data into entity instances and encapsulating the instances into entity envelope documents. The code will run as-is, but you will need to customize the code to meet the needs of your application.
- Version Translator Module: A code template for converting between different versions of a model. For example, if you add a new entity type or a new entity property, you can use a converter module to easily upgrade your entity instances to the new model.

You can generate the following additional artifacts using Entity Services. The input in all cases is a model descriptor. You can extend or customize any or all of these artifacts, if needed, but they all deliver value to your application as-is.

- Model Schema: An XML schema derived from the model. Useful for validating entity instances. For example, when harmonizing source data with your model, you can use schema validation to ensure your envelope documents contain correct entity instances.
- Template Driven Extraction Template: A TDE template that can be used to generate views of your instance data as rows or triples. If you deploy the template, you can use interfaces such as SQL, SPARQL, and the Optic API to query your instances.
- Query Options: A set of query options usable with the Search API and the REST, Java, and Node.js Client APIs. For example the options define a constraint for each required property of an entity type and limit search results to returning just the canonical instance data from an envelope document.
- Database Configuration: A database configuration file compatible with `ml-gradle` that can be used to create indexes and lexicons based on your entity type definitions. You can easily extract the configuration to use with the REST Management API rather than `ml-gradle`.

For more details, see “Generating Code and Other Artifacts” on page 107.

1.4 Next Steps

Use the following suggestions to continue learning about Entity Services:

- Walk through a simple example of creating a model, harmonizing data, creating envelope documents, and searching entity instances. See “Getting Started With Entity Services” on page 19.
- Learn about creating model descriptors. See “Creating and Managing Models” on page 57.
- Learn about creating entity instances from a model. See “Managing Entity Instances” on page 149.
- Learn more about the application code, schemas, and other configuration artifacts that you can generate from a model using Entity Services. See “Generating Code and Other Artifacts” on page 107.
- Explore several end to end examples built with Entity Services. See “Exploring the Entity Services Open-Source Examples” on page 15.

1.5 Exploring the Entity Services Open-Source Examples

The Entity Services library is automatically installed when you install MarkLogic Server. The library isno longer being maintained as an open source project on GitHub. The GitHub project does contain several examples, which you recommend you download and review.

The examples in this guide are simple ones based on data from the GitHub examples, but they are independent of the GitHub examples. You might still wish to explore the GitHub examples because they illustrate end-to-end integration of Entity Services with other MarkLogic tools and interfaces.

The example directory of the project can be found at the following URL:

http://github.com/marklogic/entity-services/tree/master/entity-services-examples

Before you can deploy and run the examples, you must create a local copy of the project. You can do this using the `git` tool (or other git client), or by downloading a zip file from GitHub. For details, see one of the following topics:

Detailed instructions for deploying and running these examples are on GitHub.

1.5.1 Downloading the Project as a ZIP File

To obtain a local copy from a ZIP file, follow these steps:

1. Navigate to the following URL in your browser: http://github.com/marklogic/entity-services. The entity-services project home page on GitHub is displayed.

2. Click the “Clone or download” dropdown. A dialog box appears.

3. Click “Download ZIP”. When prompted, choose a location in which to save the ZIP file and click Save.

4. Unzip the download file to a folder of your choice. By default, this creates a folder named `entity-services-`*`branch`*. For example, you will have a directory named `entity-services-master` if you downloaded the “master” branch.

5. Change directory into `entity-services-`*`branch`*`/entity-services-examples`.

6. Follow the instructions on this page to configure, deploy, and run the examples:

   http://github.com/marklogic/entity-services/blob/master/entity-services-examples/README.md

1.6 Security Considerations

No special security privileges or roles are needed to use the Entity Services API.

The entity envelope documents, code modules, schemas, and other artifacts you generate when using the Entity Services API are generic and can be secured using the same mechanisms as other documents and modules. For example, you should use document permissions to manage access to your envelope documents and persisted model descriptor.

Special privileges might be required to deploy some of the generated artifacts. For example, the user who installs generated code modules must have permission to insert into modules database. Similarly, the user who installs a TDE template created using Entity Services requires the `tde-admin` role or equivalent privileges, as when installing any other template.

2.0 Getting Started With Entity Services

This chapter walks through a very simple Entity Services example of creating a model, creating entity instances from source data, and querying the model and instances. Choose either the XQuery walkthrough or the Server-Side JavaScript walkthrough.

2.1 Before You Begin

All the exercises in this section use the Query Console browser application to evaluate code on MarkLogic Server. You can launch Query Console by navigating to port 8000 of a host running MarkLogic.

For example, if MarkLogic is installed on localhost, launch Query Console by opening the following location in your browser:

```
http://localhost:8000
```

To use Query Console, you must have the `qconsole-user` role or equivalent privileges. You can learn more about Query Console in the *Query Console User Guide*.

> **Note:** You do not require special security privileges to use the Entity Services API. However, some exercises in this chapter involve deploying application code to MarkLogic, so you should log into Query Console as a user with the admin role or equivalent privileges.

Some exercises in this chapter save generated code and configuration artifacts to the local filesystem on the host where MarkLogic is installed, and later read them back. You can choose any directory, but the directory must be readable and writeable by MarkLogic and by you. The examples use the variable *`ARTIFACT_DIR`* to represent this directory in the instructions.

2.2 Optional: Create a Content Database

You can use any database for the exercises in this chapter. However, if you would like to isolate this work from the rest of your environment, you can use the procedure in this section to create a new content database named “es-gs”, with one forest of the same name attached to it.

The following procedure uses the XQuery Admin API to create a database and a forest, and then attach the forest to the database. You could also use the Admin Interface or the REST Management API.

1. Navigate to Query Console in your browser. For example, if MarkLogic is installed on localhost, navigate to the following URL:

   http://localhost:8000/qconsole

2. When prompted for login credentials, login as a user with admin privileges.

3. Add a new query to the workspace by clicking on the "+" button on the query editor.

4. Select XQuery in the Query Type dropdown.

5. Copy and paste the following code into the new query. This code creates a forest and a database, and then attaches the forest to the database.

```
(: create a database:)
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";
admin:save-configuration(
  admin:database-create(admin:get-configuration(),
    "es-gs", xdmp:database("Security"), xdmp:database("Schemas")));

(: create a forest :)
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";
admin:save-configuration(
  admin:forest-create(admin:get-configuration(),
    "es-gs", xdmp:host(), ()));

(: attach the forest to the database :)
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";
admin:save-configuration(
  admin:database-attach-forest(admin:get-configuration(),
    xdmp:database("es-gs"), xdmp:forest("es-gs")));
```

6. Click the Run button. A database named "es-gs" is created.

7. Optionally, confirm the existence of the new database by browsing to the Admin Interface. For example, browse to http://localhost:8001 and observe "es-gs" in the list of databases.

2.3 Getting Started Using XQuery

This section uses XQuery and XML to introduce the Entity Services APIs. If you prefer to use Server-Side JavaScript, see "Getting Started Using JavaScript" on page 37. You can also use JSON with XQuery and XML with JavaScript, but these combinations are not illustrated here.

- Stage the Source Data

- Create a Model Descriptor
- Create a Model
- Create and Deploy an Instance Converter
- Create Entity Instances
- Query the Data
- Query the Model

2.3.1 Stage the Source Data

This exercise ingests the raw source data from which we will create entity instances. One benefit of Entity Services is that you do not have to model your data up front. You can load your data as-is and use it in your application, and then incrementally model your entities.

You usually create entity instances from XML or JSON data. The raw data in this example is 2 XML documents and a JSON document. Each document contains information about a person, such as first name and last name. Each person document also includes a unique persond identifier.

Use the following procedure to load the raw source documents into your content database. The newly created documents are put into a collection named “raw” so we can easily reference them later.

1. Navigate to Query Console in your browser. For example, if MarkLogic is installed on localhost, navigate to the following URL:

   http://localhost:8000/qconsole

2. Add a new query to the workspace by clicking on the “+” button on the query editor.

3. Select XQuery in the Query Type dropdown.

4. Select your content database from the Database dropdown.

5. Copy and paste the following code into the new query.

```
(: Stage raw source in the form of 2 XML and 1 JSON document :)
xquery version "1.0-ml";
import module namespace es = "http://marklogic.com/entity-services"
    at "/MarkLogic/entity-services/entity-services.xqy";

(: Synthesize source data in memory. Normally, this would come
 : from an external source. :)
let $source-data := (
  <person>
    <pid>1234</pid>
    <given>George</given>
    <family>Washington</family>
```

```
  </person>,
  xdmp:unquote('
    {"pid": 2345,
     "given": "Martha",
     "family": "Washington"}
  ')/node(),
  <person>
    <pid>3456</pid>
    <given>Alexander</given>
    <family>Hamilton</family>
  </person>
)
for $source in $source-data return
  let $uri-suffix :=
    typeswitch ($source)
    case element() return ".xml"
    case object-node() return ".json"
    default return ()
  return xdmp:document-insert(
    fn:concat('/es-gs/raw/', $source/pid, $uri-suffix),
    $source,
    <options xmlns="xdmp:document-insert">
      <collections>
        <collection>raw</collection>
      </collections>
    </options>
  )
```

6. Click the Run button. Three documents are created in the database.

7. Optionally, click the Explore button and observe that the following documents were created in the "raw" collection.

   /es-gs/raw/1234.xml
   /es-gs/raw/2345.json
   /es-gs/raw/3456.xml

2.3.2 Create a Model Descriptor

You define the entity types, attributes, and relationships of your model in an XML or JSON *model descriptor*. The model descriptor is the foundation for the model. Model descriptors are discussed in detail in "Creating and Managing Models" on page 57.

The model descriptor in this example is based on the `Person` example from the Entity Services examples on GitHub. For more details about the original example, see "Exploring the Entity Services Open-Source Examples" on page 15.

This exercise saves an XML model descriptor as a file on the filesystem. Discussion of the descriptor follows the procedure. For an equivalent JSON example, see “Create a Model Descriptor” on page 39.

1. Choose a filesystem directory on your MarkLogic host to hold the model descriptor file. The exercises in this chapter use *ARTIFACT_DIR* to represent this location.

2. Create a text file named `person-desc.xml` in *ARTIFACT_DIR* with the following contents.

```
<es:model xmlns:es="http://marklogic.com/entity-services">
  <es:info>
    <es:title>Person</es:title>
    <es:version>1.0.0</es:version>
    <es:base-uri>http://example.org/example-person/</es:base-uri>
    <es:description>
      A model of a person, to demonstrate several extractions
    </es:description>
  </es:info>
  <es:definitions>
    <Person>
      <es:properties>
        <id><es:datatype>string</es:datatype></id>
        <firstName><es:datatype>string</es:datatype></firstName>
        <lastName><es:datatype>string</es:datatype></lastName>
        <fullName><es:datatype>string</es:datatype></fullName>
        <friends>
          <es:datatype>array</es:datatype>
          <es:items><es:ref>#/definitions/Person</es:ref></es:items>
        </friends>
      </es:properties>
      <es:primary-key>id</es:primary-key>
      <es:required>firstName</es:required>
      <es:required>lastName</es:required>
      <es:required>fullName</es:required>
    </Person>
  </es:definitions>
</es:model>
```

3. Set the permissions on *ARTIFACT_DIR* and the newly created file so that MarkLogic can read the file.

You now have a file named *ARTIFACT_DIR*`/person-desc.xml` that contains the `Person` model descriptor.

We stored the model on the filesystem because this most closely resembles a real development cycle, in which an important project artificat like the model descriptor is under source control.

The descriptor defines a single entity type named `Person`. A `Person` entity instance contains string-valued properties named `id`, `firstName`, `lastName`, `fullName` and a list-valued property named `friends`.

```
<Person>
  <es:properties>
    <id><es:datatype>string</es:datatype></id>
    <firstName><es:datatype>string</es:datatype></firstName>
    <lastName><es:datatype>string</es:datatype></lastName>
    <fullName><es:datatype>string</es:datatype></fullName>
    <friends>
      <es:datatype>array</es:datatype>
      <es:items><es:ref>#/definitions/Person</es:ref></es:items>
    </friends>

    ...
```

The `friends` property is a list (array) of references to other `Person` entities. Since the reference to `Person` appears in the same descriptor in which `Person` is defined, it is a “local reference”. Entity Services knows the “shape” of the referenced entity type when generating code from a `Person` model. You can also reference entity types defined elsewhere.

The `firstName`, `lastName`, and `fullName` properties must all be present in every `Person` entity instance because these properties are explicitly flagged as required through the use of `<es:required/>`:

```
<es:required>firstName</es:required>
<es:required>lastName</es:required>
<es:required>fullName</es:required>
```

The `id` property is implicitly required because it is identified as the primary key for a `Person`:

```
<es:primary-key>id</es:primary-key>
```

The primary key is a unique identifier for an entity instance. You are not required to define a primary key, but the existence of a primary key facilitates other Entity Services features; for details, see “Identifying the Primary Key Entity Property” on page 67.

Since the `friends` property is neither a primary key nor an explicitly required property, it is optional. That is, you can create entities that do not include a `friends` property.

You can also flag properties with other characteristics, such as whether or not a property should be indexed for efficient search. For more details, see “Writing a Model Descriptor” on page 59.

2.3.3 Create a Model

Inserting an XML or JSON model descriptor document into the special collection `http://marklogic.com/entity-services/models` tells MarkLogic the document is part of an Entity Services model. Membership in this collection causes MarkLogic to generate semantic triples that define the model.

We "authored" a model descriptor in "Create a Model Descriptor" on page 22. The following procedure covers the validation and persistence steps that create the model. An explanation of the code follows the procedure.

1. Open Query Console in your browser if you do not already have it open.

2. Add a new query to the workspace by clicking on the "+" button on the query editor.

3. Select XQuery in the Query Type dropdown.

4. Select your content database from the Database dropdown.

5. Copy and paste the following code into the new query. This code creates a model from a descriptor.

```
(: Create a model. :)
xquery version "1.0-ml";
import module namespace es = "http://marklogic.com/entity-services"
  at "/MarkLogic/entity-services/entity-services.xqy";

let $ARTIFACT_DIR := '/space/es/gs/'
let $desc := xdmp:document-get(
  fn:concat($ARTIFACT_DIR, 'person-desc.xml'))
let $validated-desc := es:model-validate($desc)
let $desc-as-json := xdmp:to-json($validated-desc)
return xdmp:document-insert(
  '/es-gs/models/person-1.0.0.json', $desc-as-json,
  <options xmlns="xdmp:document-insert">
    <collections>{
      <collection>http://marklogic.com/entity-services/models</collection>,
      for $coll in xdmp:default-collections()
      return <collection>{$coll}</collection>
    }</collections>
  </options>
)
```

6. Change the value of the *ARTIFACT_DIR* variable to the directory where you saved the model descriptor in "Create a Model Descriptor" on page 22. Include the trailing directory separator in the pathname.

7. Click the Run button. A model is created. The descriptor is persisted as a document with the URI `/es-gs/models/person-1.0.0.json`.

   If the query is unable to open the input model descriptor file, check the permissions on the directory and file.

8. Optionally, click the Explore button at the top of the query editor to view the JSON version of the descriptor.

The first step is to validate the descriptor. An invalid descriptor will produce an invalid model. Validation introduces overhead, but an invalid descriptor will produce an invalid model, so validation is recommended during development.

```
let $desc := xdmp:document-get(
  fn:concat($ARTIFACT_DIR, 'person-desc.xml'))
let $validated-desc := es:model-validate($desc)
```

The function `es:model-validate` returns a `json:object` representation of the descriptor. A `json:object` is a special kind of `map:map`. This is the form expected by Entity Services API functions that operate on the model, but it is not the proper form for creating a model. Instead, you must persist an XML or JSON descriptor.

If you persist a descriptor as XML, then you must use `es:model-validate` or `es:model-from-xml` to convert it to the `map:map` form if you extract it from the database to pass to an Entity Services function. If you persist the descriptor as JSON, then subsequent conversion is not necessary. Therefore, this example persists a JSON version of the original XML descriptor.

The function `xdmp:to-json` converts the `json:object` created by `es:model-validate` into a JSON object-node that represents the JSON version of our XML descriptor. For example:

```
let $desc-as-json := xdmp:to-json($validated-desc)
```

Finally, we insert the descriptor into the database as part of the special Entity Services collection to create the model. The following document insertion adds the Entity Services collection to any default collections associated with the user performing the insertion.

```
xdmp:document-insert(
  '/es-gs/models/person-1.0.0.json', $model-as-json,
  <options xmlns="xdmp:document-insert">
    <collections>{
      <collection>http://marklogic.com/entity-services/models</collection>,
      for $coll in xdmp:default-collections()
      return <collection>{$coll}</collection>
    }</collections>
  </options>
)
```

2.3.4 Create and Deploy an Instance Converter

An instance converter is a library module containing code for transforming your raw source data into entity instances that conform to your model. You can use the Entity Services API to generate a baseline converter, and then customize it to meet the requirements of your application.

This section walks through deploying a converter module in the following steps:

- Generate the Default Converter Module
- Customize the Converter Module
- Deploy the Converter Module

2.3.4.1 Generate the Default Converter Module

This exercise creates an instance converter module template using the `es:instance-converter-generate` function. An explanation of the code follows the procedure.

1. Open Query Console in your browser if you do not already have it open.
2. Add a new query to the workspace by clicking on the "+" button on the query editor.
3. Select XQuery in the Query Type dropdown.
4. Select your content database from the Database dropdown.
5. Copy and paste the following code into the new query. This code generates the instance converter module and saves it to the filesystem.

```
(: Create an instance converter and save it to a file :)
xquery version "1.0-ml";
import module namespace es = "http://marklogic.com/entity-services"
  at "/MarkLogic/entity-services/entity-services.xqy";

let $desc := fn:doc('/es-gs/models/person-1.0.0.json')
let $ARTIFACT_DIR := '/space/es/gs/'     (: MODIFY THIS VALUE :)
return xdmp:save(
  fn:concat($ARTIFACT_DIR, 'person-1.0.0-conv.xqy'),
  es:instance-converter-generate($desc)
)
```

6. Change the value of `$ARTIFACT_DIR` to a directory on your MarkLogic host where the generated code can be saved. Include the trailing directory separator in the pathname.

   The directory must be readable and writable by MarkLogic.

7. Click the Run button. The file `ARTIFACT_DIR/person-1.0.0-conv.xqy` is created.

8. Optionally, go to *ARTIFACT_DIR* and review the generated code. In the next section, we will modify this code.

Though the generated code is runnable as-is, you will need to customize the code to match the characteristics of your source data and the requirements of your application. The generated code contains extensive comments to assist you with customization.

We could insert the converter module directly into the modules database to which it will eventually be deployed. However, the converter is an important project artifact, so you would normally save it to a file and place it under source control before proceeding with customizations.

The generated module defines the following externally visible functions, plus some private helper functions. The namespace prefix defined for the module is derived from the model title.

- `person:extract-instance-Person` - Create a `Person` instance from raw source data. The returned instance is a `json:object` (`map:map`). You are expected to customize this function to harmonize your source data with your model.
- `person:instance-to-envelope` - Convert an entity instance into an XML or JSON envelope document that encapsulates the instance and the original source. Most applications will use this function as-is, but you might customize it if you include additional data in the envelope.
- `person:instance-to-canonical` - Convert the `map:map` representation of an instance into its canonical XML or JSON representation. You will not usually need to customize this function or call it directly; it exists for use by the generated `instance-to-envelope` function.

For more details, see “Creating an Instance Converter Module” on page 110.

2.3.4.2 Customize the Converter Module

The converter module generated by Entity Services implements a *modeltitle*`:extract-instance-`*T* function for each entity type *T* defined in the descriptor. In our example, the converter module implements a `person:extract-instance-Person` function.

The default implementation of an instance converter assumes the source data has the same “shape” as a `Person` entity. However, our source data has `pid`, `given`, and `family` properties instead of `id`, `firstName`, `lastName`, and `fullName`. You must modify `person:extract-instance-Person` to do the following:

- Extract `id` from `pid`
- Extract `firstName` from `given`
- Extract `lastName` from `family`
- Synthesize `fullName` by concatenating `given` and `family`

Production applications can require many other types of customizations. For example, you might need to normalize a date value, perform a more sophisticated type conversion, or extract the value of an entity property from somewhere other than the source data.

Use the following procedure to customize the instance extraction code as described. A discussion of the code follows the procedure.

1. Confirm you have read and write permissions on *ARTIFACT_DIR*/person-1.0.0-conv.xqy. If not, set the permissions accordingly. The file must also be readable by MarkLogic.

2. Open *ARTIFACT_DIR*/person-1.0.0-conv.xqy in the text editor of your choice.

3. Locate the section of person:extract-instance-Person that prepares the value of the id, firstName, lastName, and fullName properties. The code should look similar to the following:

```
let $id  :=       $source-node/id ! xs:string(.)
let $firstName := $source-node/firstName ! xs:string(.)
let $lastName  := $source-node/lastName ! xs:string(.)
let $fullName  := $source-node/fullName ! xs:string(.)
```

4. Replace these lines with the following code. The bold text highlights the changes.

```
let $id  :=       $source-node/pid ! xs:string(.)
let $firstName := $source-node/given ! xs:string(.)
let $lastName  := $source-node/family ! xs:string(.)
let $fullName  := fn:concat($firstName, " ", $lastName) ! xs:string(.)
```

5. Save your changes.

Recall that the Person entity type has id, firstName, lastName, fullName, and friends properties. The default implementation of person:extract-instance-Person assumes the source data contains the same properties. For example, the default implementation includes the following code:

```
let $id  := $source-node/id ! xs:string(.)
let $firstName := $source-node/firstName ! xs:string(.)
let $lastName := $source-node/lastName ! xs:string(.)
let $fullName := $source-node/fullName ! xs:string(.)
```

Our customization changes the names of the source fields to match our source data, and derives the fullName property from the given and family source values. The modified portions are shown in bold, below.

```
let $id  :=       $source-node/pid ! xs:string(.)
let $firstName := $source-node/given ! xs:string(.)
let $lastName  := $source-node/family ! xs:string(.)
let $fullName  := fn:concat($firstName, " ", $lastName) ! xs:string(.)
```

2.3.4.3 Deploy the Converter Module

Like any application code, the converter module must be deployed to MarkLogic before you can use it. Best practice is to install it in the modules database of your App Server. Our example uses the pre-defined App Server on port 8000, which is configured to use the Modules database.

The following procedure uses XQuery to install the customized converter module into the Modules database. You could also use Server-Side JavaScript or the REST, Java, or Node.js Client APIs for this task.

1. Open Query Console in your browser if you do not already have it open.

2. Add a new query to the workspace by clicking on the "+" button on the query editor.

3. Select XQuery in the Query Type dropdown.

4. Select the "Modules" database from the Database dropdown.

5. Copy and paste the following code into the new query. This code saves the instance converter module to the database.

```
xquery version "1.0-ml";
let $ARTIFACT_DIR := '/space/es/gs/'     (: MODIFY THIS VALUE :)
return xdmp:document-load(
  fn:concat($ARTIFACT_DIR, 'person-1.0.0-conv.xqy'),
  <options xmlns="xdmp:document-load">
    <uri>/es-gs/person-1.0.0-conv.xqy</uri>
  </options>
)
```

6. Modify the value of *$ARTIFACT_DIR* to the directory where you previously saved the converter module. Include the trailing directory separator in the pathname.

7. Click the Run button. The converter module is inserted into the Modules database.

8. Optionally, click the Explore button to confirm the presence of the module in the database.

2.3.5 Create Entity Instances

An envelope document is the recommended way to persist and interact with entity instances in MarkLogic. An envelope document encapsulates an entity instance with model metadata and the original source. Storing the logical aspects of an entity (canonical instance representation, metadata, source) in one physical document facilitates managing, searching, retrieving, indexing, and securing your data.

An envelope document enables your application to query data as harmonized instances, but still recover the raw source when needed. You can generate either XML or JSON envelope documents.

You can use the `person:instance-to-envelope` function in the converter module to create entity envelope documents. The input is an instance created by calling `person:extract-instance-Person`. If you do not explicitly specify an envelope format of "xml" or "json", the function generates an XML envelope.

Use the following procedure to create XML envelope documents from the source documents loaded in "Stage the Source Data" on page 21. Discussion of the code follows the procedure.

1. Open Query Console in your browser if you do not already have it open.

2. Add a new query to the workspace by clicking on the "+" button on the query editor.

3. Select XQuery in the Query Type dropdown.

4. Select your content database from the Database dropdown.

5. Copy and paste the following code into the new query. This code creates a `Person` entity envelope XML document from each source document.

```
(: Create envelope documents from raw source documents :)
xquery version "1.0-ml";
import module namespace es = "http://marklogic.com/entity-services"
    at "/MarkLogic/entity-services/entity-services.xqy";
import module namespace person =
    "http://example.org/example-person/Person-1.0.0"
    at "/es-gs/person-1.0.0-conv.xqy";

for $source in fn:collection('raw') return
  let $instance := person:extract-instance-Person($source)
  let $uri :=
    fn:concat('/es-gs/env/', map:get($instance, 'id'), '.xml')
  return xdmp:document-insert(
    $uri,
    person:instance-to-envelope($instance, "xml"),
    <options xmlns="xdmp:document-insert">
      <collections>
        <collection>person-envelopes</collection>
      </collections>
    </options>
  )
```

6. Click the Run button. The following envelope documents are created in your content database:

   /es-gs/env/1234.xml
   /es-gs/env/2345.xml
   /es-gs/env/3456.xml

7. Optionally, click the Explore button to confirm creation of the envelope documents.

An envelope document can be either XML or JSON. This exercise uses XML envelopes. An XML envelope has the following form. The `es:attachments` portion of the envelope holds the raw source data.

```
<es:envelope xmlns:es="http://marklogic.com/entity-services">
  <es:instance>
    <es:info>metadata from info section of descriptor</es:info>
    ...instance canonical XML..
  </es:instance>
  <es:attachments>
    source data
  </es:attachments>
</es:envelope>
```

The equivalent JSON envelope, generated by passing "json" as the second parameter of `person:instance-to-envelope`, has the following form:

```
{ "envelope": {
  "instance": {
    "info": { ...metadata from info section of descriptor... },
    ...instance canonical JSON...
  },
  "attachments": [ ...source data... ]
}}
```

Except when constructing path expressions, you do not usually have to be aware of the internal structure of an envelope document because the Entity Services API includes functions for extracting an instance or the attachments from an envelope document handle it for you. For details, see “Extracting an Entity Instance from an Envelope Document” on page 161 and “Extracting the Original Source from an Envelope Document” on page 164.

You create an envelope document for some entity type *T* and envelope format *F* using the `extract-instance-T` and `instance-to-envelope` functions of the instance converter. For example:

```
(: creating an XML envelope :)
modeltitle:instance-to-envelope(
  modeltitle:extract-instance-T($source), "xml")

(: creating a JSON envelope :)
modeltitle:instance-to-envelope(
  modeltitle:extract-instance-T($source), "json")
```

For example, the sample code does the following to create a `Person` entity XML envelope:

```
let $instance := person:extract-instance-Person($source)
...
return xdmp:document-insert(
  $uri,
  person:instance-to-envelope($instance, "xml"),
  ...)
```

Inside `person:instance-to-envelope`, the `person:instance-to-canonical` function is called to create the Person entity embedded inside `es:envelope/es:instance`.

The table below illustrates the progression from raw data to XML envelope document, through use of the instance converter module functions.

Operation	Result

`ingest raw source`	`<person>`
	`  <pid>1234</pid>`
	`  <given>George</given>`
	`  <family>Washington</family>`
	`</person>`

`extract-instance-Person($ source)`
 	
`input: raw source`
`output: a map:map (json:object), shown here serialized as JSON`
	`{"$attachments": "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n<person>\n <pid>1234</pid>\n  <given>George</given>\n <family>Washington</family>\n</person>",`
	`  "$type": "Person",`
	`  "id": "1234",`
	`  "firstName": "George",`
	`  "lastName": "Washington",`
	`  "fullName": "George Washington"`
	`}`

Operation	Result

`instance-to-canonical($instance, "xml")`
 	
`input: instance map:map`
`output: XML elem`
	`<Person>`
	`  <id>1234</id>`
	`  <firstName>George</firstName>`
	`  <lastName>Washington</lastName>`
	`  <fullName>George Washington</fullName>`
	`</Person>`

`instance-to-envelope($instance, "xml")`
 	
`input: instance map:map`
`output: XML envelope doc`
	`<es:envelope`
	`  xmlns:es="http://marklogic.com/entity-services">`
	`  <es:instance>`
	`  <es:info>`
	`  <es:title>Person</es:title>`
	`  <es:version>1.0.0</es:version>`
	`  </es:info>`
	`  <Person>`
	`  <id>1234</id>`
	`  <firstName>George</firstName>`
	`  <lastName>Washington</lastName>`
	`  <fullName>George Washington</fullName>`
	`  </Person>`
	`  </es:instance>`
	`  <es:attachments>`
	`  <person>`
	`  <pid>1234</pid>`
	`  <given>George</given>`
	`  <family>Washington</family>`
	`  </person>`
	`  </es:attachments>`
	`</es:envelope>`

The following is an equivalent JSON envelope, generated by calling `instance-to-envelope($instance, "json")`:

```
{ "envelope": {
  "instance": {
    "info": {
      "title":"Person",
      "version":"1.0.0"
    },
    "Person": {
      "id":"2345",
      "firstName":"Martha",
      "lastName":"Washington",
      "fullName":"Martha Washington"}
    },
    "attachments":[

"<person><pid>2345</pid><given>Martha</given><family>Washington</family></pers
on>"
    ]
}}
```

Note that the source data in the attachments is represented as a string if it does not match the envelope data format. For example, in the above JSON envelope, the source attachment is a string, rather than an XML node. This has implications for extracting the source from the envelope as a node; see the example in "Query the Data" on page 51.

2.3.6 Query the Data

This section illustrates one way to search your entity instance data using the cts:search XQuery function. You can also use other MarkLogic document search APIs, search your instances as row data, or use semantic search. The Entity Services API includes tools to facilitate all these forms of search. For details, see "Querying a Model or Entity Instances" on page 169.

The following example uses the XQuery `cts:query` API to find all Person entities with a `lastName` property of Washington, and then emits the original source from which the entity was derived.

1. Open Query Console in your browser if you do not already have it open.

2. Add a new query to the workspace by clicking on the "+" button on the query editor.

3. Select XQuery in the Query Type dropdown.

4. Select your content database from the Database dropdown.

5. Copy and paste the following code into the new query. The code matches documents in the `person-envelopes` collection where the `lastName` element has the value "washington", and then returns the original source data from the envelope.

```
xquery version "1.0-ml";
import module namespace es = "http://marklogic.com/entity-services"
  at "/MarkLogic/entity-services/entity-services.xqy";

(: match all envelopes containing an entity instances with
 : a lastName property value of 'washington' :)
let $matches := cts:search(
  fn:collection('person-envelopes'),
  cts:element-query(
    fn:QName('http://marklogic.com/entity-services', 'instance'),
    cts:element-value-query(xs:QName('lastName'), 'washington')
  ))
(: extract the original source, as a node :)
for $attachment in $matches/es:envelope/es:attachments/node()
return typeswitch ($attachment)
  case element() return $attachment
  case text() return xdmp:from-json-string($attachment)
  default return ()
```

6. Click the Run button. The query returns a JSON node and an XML node similar to the following:

```
{ "pid":2345,
  "given":"Martha",
  "family":"Washington" }

<person xmlns:es="http://marklogic.com/entity-services">
  <pid>1234</pid>
  <given>George</given>
  <family>Washington</family>
</person>
```

The search matches two entity instances, one extracted from JSON source and one extracted from XML source, so final query results are one JSON node and one XML node.

The search is limited to the envelope documents by specifying the `person-envelopes` collection. A container query (`cts:element-query`) further constrains the search to occurrences within the `es:instance portion` of an envelope document. Finally, a `cts:element-value-query` is used to match envelopes where the `lastName` property value is “washington”.

```
cts:search(
  fn:collection('person-envelopes'),
  cts:element-query(
    fn:QName('http://marklogic.com/entity-services', 'instance'),
    cts:element-value-query(xs:QName('lastName'), 'washington')
  ))
```

The container query ensures the search will not find matches in any part of the envelope document except the entity instance. You could similarly search just the `es:attachments`, but remember that you cannot perform a structured search on JSON source in the attachments because it is stored in the envelope document as a string.

Notice that the example code can return the original XML source data directly out of the envelope document, but the original JSON document must be converted from a string to a JSON node using `xdmp:from-json-string`, if you want to return it as a node.

2.3.7 Query the Model

When you created a model in “Create a Model” on page 25, MarkLogic automatically generated some facts from the persisted descriptor, as semantic triples. These facts (and any additional facts you add) define the model and enable semantic queries against the model.

For example, you can use a SPARQL query to discover what entity types are defined by a model, what properties are required in an entity instance of a particular type, or the datatype of a particular entity type property. For more details, see “Querying a Model or Entity Instances” on page 169.

The following procedure uses a SPARQL query to generate a list of all the required properties of an instance of the `Person` entity type:

1. Open Query Console in your browser if you do not already have it open.

2. Add a new query to the workspace by clicking on the "+" button on the query editor.

3. Select SPARQL Query in the Query Type dropdown.

4. Select your content database from the Database dropdown.

5. Copy and paste the following code into the new query. This code retrieves the names of all required properties of a `Person` entity instance.

```
prefix es:<http://marklogic.com/entity-services#>
select ?ptitle
where {
  ?x a es:EntityType;
       es:title "Person";
       es:property ?property .
  ?property a es:RequiredProperty;
              es:title ?ptitle
}
```

6. Click the Run button. The query results are displayed as a table.

You should see results similar to the following:

```
ptitle
"lastName"
"fullName"
"firstName"
```

You can also use the SQL and Optic APIs to query your model and entities as rows if you install an Entity Services generated TDE template based on your model. For more details and examples, see "Querying a Model or Entity Instances" on page 169. To learn more about Semantics in MarkLogic Server, see the *Semantics Developer's Guide*.

2.4 Getting Started Using JavaScript

This section uses Server-Side JavaScript and JSON to introduce the Entity Services APIs. If you prefer to use XQuery, see "Getting Started Using XQuery" on page 20. You can also use JSON with XQuery and XML with JavaScript, but these combinations are not illustrated here.

- Stage the Source Data
- Create a Model Descriptor
- Create a Model

- Create and Deploy an Instance Converter
- Create Entity Instances
- Query the Data
- Query the Model

2.4.1 Stage the Source Data

This exercise ingests the raw source data from which we will create entity instances. One benefit of Entity Services is that you do not have to model your data up front. You can load your data as-is and use it in your application, and then incrementally model your entities.

You usually create entity instances from XML or JSON data. The raw data in this example is 2 XML documents and a JSON document. Each document contains information about a person, such as first name and last name. Each person document also includes a unique persond identifier.

Use the following procedure to load the raw source documents into your content database. The newly created documents are put into a collection named “raw” so we can easily reference them later.

1. Navigate to Query Console in your browser. For example, if MarkLogic is installed on localhost, navigate to the following URL:

   http://localhost:8000/qconsole

2. Add a new query to the workspace by clicking on the “+” button on the query editor.

3. Select JavaScript in the Query Type dropdown.

4. Select your content database from the Database dropdown.

5. Copy and paste the following code into the new query.

```
'use strict';
declareUpdate();

// Synthesize source data in memory. This would normally come
// from an external source.
const sourceData = [
  fn.head(xdmp.unquote(
    '<person>' +
      '<pid>1234</pid>' +
      '<given>George</given>' +
      '<family>Washington</family>' +
    '</person>')),
  {pid: 2345,
   given: 'Martha',
   family: 'Washington'},
```

```
    fn.head(xdmp.unquote(
      '<person>' +
        '<pid>3456</pid>' +
        '<given>Alexander</given>' +
        '<family>Hamilton</family>' +
      '</person>'))
];

// Insert each source item into the db as an XML or JSON doc.
sourceData.forEach(function(source) {
  let uri = '/es-gs/raw/';
  if (source instanceof Document) {
    // XML doc created by xdmp.unquote
    uri += source.xpath('/node()/pid/data()') + '.xml';
  } else if (source instanceof Object) {
    uri += source.pid + '.json';
  }
  xdmp.documentInsert(uri, source, {collections: ['raw']});
});
```

6. Click the Run button. Three documents are created in the database.

7. Optionally, click the Explore button and observe that the following documents were created in the "raw" collection.

   /es-gs/raw/1234.xml
   /es-gs/raw/2345.json
   /es-gs/raw/3456.xml

The `sourceData` array, above, creates raw data in a very artifical way in order to have a self-contained example. Your source data will normally come from an external source, such as files on the file system, an HTTP request payload, or an mlcp job.

Part of this artificiality is the use of `xdmp.unquote` as quick way to create an XML node from a literal. You would normally use NodeBuilder to create in-memory XML documents from Server-Side JavaScript.

2.4.2 Create a Model Descriptor

You define the entity types, entity type properties, and relationships of your model in an XML or JSON *model descriptor*. The model descriptor is the staring point for creating a model. Model descriptors are discussed in detail in "Creating and Managing Models" on page 57.

The model descriptor in this example is based on the `Person` example from the Entity Services examples on GitHub. For more details about the original example, see "Exploring the Entity Services Open-Source Examples" on page 15.

This exercise saves a JSON model descriptor as a file on the filesystem. Discussion of the descriptor follows the procedure.

1. Choose a filesystem directory on your MarkLogic host to hold the model descriptor file. The exercises in this chapter use *ARTIFACT_DIR* to represent this location.

2. Create a text file named file `person-desc.json` in *ARTIFACT_DIR* with the following contents.

```
{ "info": {
    "title": "Person",
    "version": "1.0.0",
    "baseUri": "http://example.org/example-person/",
    "description":
      "A model of a person, to demonstrate several extractions"
  },
  "definitions": {
    "Person": {
      "properties": {
        "id": {"datatype": "string"},
        "firstName": {"datatype": "string"},
        "lastName": {"datatype": "string"},
        "fullName": {"datatype": "string"},
        "friends": {
          "datatype": "array",
          "items": {"$ref": "#/definitions/Person"
        }
      }},
      "primaryKey": "id",
      "required": ["firstName", "lastName", "fullName"]
    }
  }
}
```

3. Set the permissions on *ARTIFACT_DIR* and the newly created file so that MarkLogic can read the file.

You now have a file named *ARTIFACT_DIR*`/person-desc.json` that contains the `Person` model descriptor. For an example of the equivalent XML descriptor, see “Create a Model Descriptor” on page 39 in the XQuery walkthrough.

We stored the model on the filesystem because this most closely resembles a real development cycle, in which an important project artifact like the model descriptor is under source control.

The descriptor defines a single entity type named `Person`. A `Person` entity instance contains string-valued properties named `id`, `firstName`, `lastName`, `fullName` and list-valued property named `friends`.

```
"Person": {
  "properties": {
```

```
        "id": {"datatype": "string"},
        "firstName": {"datatype": "string"},
        "lastName": {"datatype": "string"},
        "fullName": {"datatype": "string"},
        "friends": {
          "datatype": "array",
          "items": {"$ref": "#/definitions/Person"
        }
    }}, ...
```

The `friends` property is a list (array) of references to other `Person` entities. Since the reference to `Person` appears in the same descriptor in which `Person` is defined, it is a "local reference". Entity Services knows the "shape" of the referenced entity type when generating code from a `Person` model. You can also reference entity types defined elsewhere.

The `firstName`, `lastName`, and `fullName` properties must be present in every Person entity instance because these properties are explicitly flagged as required through the `required` descriptor property:

```
"required": ["firstName", "lastName", "fullName"]
```

The `id` property is implicitly required because it is identified as the primary key for a `Person`:

```
"primaryKey":"id"
```

The primary key is a unique identifier for an entity instance. You are not required to define a primary key, but the existence of a primary key facilitates other Entity Services features; for details, see "Identifying the Primary Key Entity Property" on page 67.

Since the `friends` property is neither a primary key nor an explicitly required property, it is optional. That is, you can create `Person` entities that do not include a `friends` property.

You can also flag properties with other characteristics, such as whether or not a property should be indexed for efficient search. For more details, see "Writing a Model Descriptor" on page 59.

2.4.3 Create a Model

Inserting an XML or JSON model descriptor document into the special collection `http://marklogic.com/entity-services/models` tells MarkLogic the document is part of an Entity Services model. Membership in this collection causes MarkLogic to generate semantic triples that define the model.

We "authored" a model descriptor in "Create a Model Descriptor" on page 39. The following procedure covers the validation and persistence steps that create the model. An explanation of the code follows the procedure.

The following procedure creates a model using the `Person` model descriptor. An explanation of the code follows the procedure.

1. Open Query Console in your browser if you do not already have it open.

2. Add a new query to the workspace by clicking on the "+" button on the query editor.

3. Select JavaScript in the Query Type dropdown.

4. Select your content database from the Database dropdown.

5. Copy and paste the following code into the new query. This code creates a model from a descriptor.

```
'use strict';
declareUpdate();
const es = require('/MarkLogic/entity-services/entity-services.xqy');

// Retrieve descriptor from filesystem
const ARTIFACT_DIR = '/space/es/gs/';      // CHANGE THIS VALUE
const desc = fn.head(
  xdmp.documentGet(ARTIFACT_DIR + 'person-desc.json'));

// Create the model
xdmp.documentInsert(
  '/es-gs/models/person-1.0.0.json', es.modelValidate(desc),
  {collections: ['http://marklogic.com/entity-services/models']}
);
```

6. Change the value of the `ARTIFACT_DIR` variable to the directory where you saved the model descriptor in "Create a Model Descriptor" on page 39. Include the trailing directory separator in the pathname.

7. Click the Run button. A model is created. The descriptor portion is persisted as a document with the URI `/es-gs/models/person-1.0.0.json`.

   If the query is unable to open the model descriptor file, check the permissions on the directory and file.

8. Optionally, click the Explore button at the top of the query editor to view the descriptor document in the database.

The model is created by persisting the descriptor as part of the collection `http://marklogic.com/entity-services/models`.

```
xdmp.documentInsert(
  '/es-gs/models/person-1.0.0.json', es.modelValidate(desc),
  {collections: ['http://marklogic.com/entity-services/models']}
);
```

The example also uses `es.modelValidate` to check the descriptor for errors before inserting it. An invalid descriptor will generate an invalid model. If the descriptor is invalid, `es.modelValidate` throws an exception. If you know your model descriptor is valid, you can skip validation. Skipping validation is faster, but validation is recommended during development.

2.4.4 Create and Deploy an Instance Converter

An instance converter is an XQuery library module containing code for transforming your raw source data into entity instances that conforms to your model. You can use the Entity Services API to generate a baseline converter, and then customize it to meet the requirements of your application.

This section walks through deploying a converter module in the following steps:

- Generate the Default Converter Module
- Customize the Converter Module
- Deploy the Converter Module

2.4.4.1 Generate the Default Converter Module

This exercise creates an instance converter module template using the `es.instanceConverterGenerate` function. An explanation of the code follows the procedure.

1. Open Query Console in your browser if you do not already have it open.
2. Add a new query to the workspace by clicking on the "+" button on the query editor.
3. Select JavaScript in the Query Type dropdown.
4. Select your content database from the Database dropdown.
5. Copy and paste the following code into the new query. This code generates the instance convert module and saves it to the filesystem.

```
'use strict';
const es = require('/MarkLogic/entity-services/entity-services.xqy');

const ARTIFACT_DIR = '/space/es/gs/';       // CHANGE THIS VALUE

const desc = cts.doc('/es-gs/models/person-1.0.0.json');
xdmp.save(
  ARTIFACT_DIR + 'person-1.0.0-conv.xqy',
  es.instanceConverterGenerate(desc)
);
```

6. Change the value of *ARTIFACT_DIR* to a directory on your MarkLogic host where the generated code can be saved. Include the trailing directory separator in the pathname.

   The directory must be readable and writable by MarkLogic.

7. Click the Run button. The file *ARTIFACT_DIR*/person-1.0.0-conv.xqy is created.

8. Optionally, go to *ARTIFACT_DIR* and review the generated code. In the next section, we will modify this code.

We could have inserted the converter module directly into the modules database to which it will eventually be deployed. However, the converter is an important project artifact, so you would normally save it to a file and place it under source control. Also, most applications will require converter customizations.

The generated code is runnable as-is, but you are expected to customize the code to match the characteristics of your source data and the requirements of your application. The generated code contains comments to assist you with customization. You will need to understand some XQuery to customize the converter for a production application.

The generated module defines the following functions. The namespace prefix defined for the module is derived from the model title.

- person:extract-instance-Person - Create a Person instance from raw source data. You are expected to customize this function to harmonize your source data with your model.
- person:instance-to-envelope - Convert an entity instance into an XML or JSON envelope document that encapsulates the instance and the original source. Most applications will use this function as-is, but you might customize if you include additional data in the envelope.
- person:instance-to-canonical - Convert the JSON object representation of an instance into its canonical XML or JSON representation. You will not usually need to customize this function or call it directly; it exists for use by the generated instance-to-envelope function.

As with any XQuery module in MarkLogic, you can use the instance converter module from Server-Side JavaScript, once you install the module. Bring the module into scope using a require statement. For example, if the module is installed in the modules database with the URI "/es-gs/person-1.0.0-conv.xqy", then use a require statement such as the following:

```
const person = require('/es-gs/person-1.0.0-conv.xqy');
```

Invoke the functions using their JavaScript-style, camel-case names. For example, in the case of the Person entity type, the module converter functions can be invoked from Server-Side JavaScript using the following names, assuming the module is represented by a variable named person, as shown in the above require statement.

```
person.extractInstancePerson
person.instanceToEnvelope
person.instanceToCanonical
```

For more details, see “Creating an Instance Converter Module” on page 110.

2.4.4.2 Customize the Converter Module

The converter module generated by Entity Services implements a *modeltitle*`:extract-instance-`*T* function for each entity type *T* defined in the descriptor. In our example, the converter module implements a `person:extract-instance-Person` function.

The default implementation of an instance converter assumes the source data has the same “shape” as a `Person` entity. However, our source data has `pid`, `given`, and `family` properties instead of `id`, `firstName`, `lastName`, and `fullName`. You must modify `person:extract-instance-Person` to do the following:

- Extract `id` from `pid`
- Extract `firstName` from `given`
- Extract `lastName` from `family`
- Synthesize `fullName` by concatenating `family` and `given`

Production applications can require many other types of customizations. For example, you might need to normalize a date value, perform a more sophisticated type conversion, or extract the value of an entity property from somewhere other than the source data.

Use the following procedure to customize the instance extraction code. A discussion of the code follows the procedure.

1. Confirm you have read and write permissions on *ARTIFACT_DIR*`/person-1.0.0-conv.xqy`. If not, set the permissions accordingly. The file must also be readable by MarkLogic.

2. Open *ARTIFACT_DIR*`/person-1.0.0-conv.xqy` in the text editor of your choice.

3. Locate the section of `person:extract-instance-Person` that sets the value of the `id`, `firstName`, `lastName`, and `fullName` properties. The code should look similar to the following:

   ```
   let $id  :=       $source-node/id ! xs:string(.)
   let $firstName := $source-node/firstName ! xs:string(.)
   let $lastName  := $source-node/lastName ! xs:string(.)
   let $fullName  := $source-node/fullName ! xs:string(.)
   ```

4. Replace these lines with the following code. The text in bold highlights the changes.

   ```
   let $id  :=       $source-node/pid ! xs:string(.)
   let $firstName := $source-node/given ! xs:string(.)
   ```

```
let $lastName  := $source-node/family ! xs:string(.)
let $fullName  := fn:concat($firstName, " ", $lastName) ! xs:string(.)
```

5. Save your changes.

Recall that the `Person` entity type has `id`, `firstName`, `lastName`, `fullName`, and `friends` properties. The default implementation of `person:extract-instance-Person` assumes the source data contains the same properties. For example, the default implementation includes the following code:

```
let $id  :=       $source-node/id ! xs:string(.)
let $firstName := $source-node/firstName ! xs:string(.)
let $lastName  := $source-node/lastName ! xs:string(.)
let $fullName  := $source-node/fullName ! xs:string(.)
```

Each of the variable declarations assumes the value of a property in the new entity instance (`$instance`) is the value of a property with the same name in the source node. Since that assumption does not match the example model, customization is required.

Our customization changes the names of the source fields to match our source data, and derives the `fullName` property value from the `given` and `family` source values. The modified portions are shown in bold, below.

```
let $id  :=       $source-node/pid ! xs:string(.)
let $firstName := $source-node/given ! xs:string(.)
let $lastName  := $source-node/family ! xs:string(.)
let $fullName  := fn:concat($firstName, " ", $lastName) ! xs:string(.)
```

2.4.4.3 Deploy the Converter Module

Like any application code, the converter module must be deployed to MarkLogic before you can use it. Best practice is to install it in the modules database of your App Server. Our example uses the pre-defined App Server on port 8000, which is configured to use the Modules database.

The following procedure uses XQuery to install the customized converter module into the Modules database. You could also use Server-Side JavaScript or the REST, Java, or Node.js Client APIs for this task.

1. Open Query Console in your browser if you do not already have it open.

2. Add a new query to the workspace by clicking on the “+” button on the query editor.

3. Select JavaScript in the Query Type dropdown.

4. Select the “Modules” database from the Database dropdown.

5. Copy and paste the following code into the new query. This code saves the instance converter module to the database.

```
// ** RUN AGAINST MODULES DB **
'use strict';
declareUpdate();

const ARTIFACT_DIR = '/space/es/gs/';       // CHANGE THIS VALUE

xdmp.documentLoad(
  ARTIFACT_DIR + 'person-1.0.0-conv.xqy',
  { uri: '/es-gs/person-1.0.0-conv.xqy' }
);
```

6. Modify the value of `ARTIFACT_DIR` to the directory where you previously saved the converter module. Include the trailing directory separator in the pathname.

7. Click the Run button. The converter module is inserted into the Modules database.

8. Optionally, click the Explore button to confirm the presence of the module in the database.

2.4.5 Create Entity Instances

An envelope document is the recommended way to persist and interact with entity instances in MarkLogic. An envelope document encapsulates an entity instance with model metadata and the original source. Storing the logical aspects of an entity (canonical instance representation, metadata, source) in one physical document facilitates managing, searching, retrieving, indexing, and securing your data.

An envelope document enables your application to query data as harmonized instances, but still recover the raw source when needed. You can generate either XML or JSON envelope documents.

You can use the `person.instanceToEnvelope` function in the converter module to create entity envelope documents. The input is an instance created by calling `person.extractInstancePerson`. If you do not explicitly specify an envelope format of “xml” or “json”, the function generates an XML envelope.

Use the following procedure to create envelope documents from the source documents loaded in “Stage the Source Data” on page 38. Discussion of the code follows the procedure.

1. Open Query Console in your browser if you do not already have it open.

2. Add a new query to the workspace by clicking on the “+” button on the query editor.

3. Select JavaScript in the Query Type dropdown.

4. Select your content database from the Database dropdown.

5. Copy and paste the following code into the new query. This code creates a `Person` entity envelope document from each source document.

```
'use strict';
declareUpdate();
const es = require('/MarkLogic/entity-services/entity-services.xqy');
const person = require('/es-gs/person-1.0.0-conv.xqy');

for (const source of fn.collection('raw')) {
  let instance = person.extractInstancePerson(source);
  let uri = '/es-gs/env/' + instance.id + '.xml';
  xdmp.documentInsert(
    uri, person.instanceToEnvelope(instance, "xml"),
    {collections: ['person-envelopes']}
  );
}
```

6. Click the Run button. The following envelope documents are created in your content database:

   /es-gs/env/1234.xml
   /es-gs/env/2345.xml
   /es-gs/env/3456.xml

7. Optionally, click the Explore button to confirm creation of the envelope documents.

An envelope document can be either XML or JSON. This exercise uses XML envelopes. An XML envelope has the following form. The `es:attachments` portion of the envelope holds the raw source data.

```
<es:envelope xmlns:es="http://marklogic.com/entity-services">
  <es:instance>
    <es:info>metadata from info section of descriptor</es:info>
    ...instance canonical XML..
  </es:instance>
  <es:attachments>
    source data
  </es:attachments>
</es:envelope>
```

The equivalent JSON envelope, generated by passing "json" as the second parameter of `person.instanceToEnvelope`, has the following form:

```
{ "envelope": {
  "instance": {
    "info": { ...metadata from info section of descriptor... },
    ...instance canonical JSON...
  },
  "attachments": [ ...source data... ]
}}
```

Except when constructing path expressions, you do not usually have to be aware of the internal structure of an envelope document because the Entity Services API includes functions for extracting an instance or the attachments from an envelope document handle it for you. For details, see "Managing Entity Instances" on page 149.

You create an envelope document for some entity type *T* using the `extractInstance`*`T`* and `instanceToEnvelope` functions of the instance converter. (These are the `extract-instance-`*`T`* and `instance-to-envelope` functions in the XQuery module.) For example:

```
modeltitle.instanceToEnvelope(
  modeltitle.extractInstanceT($source))
```

For example, the sample code does the following to create a Person entity envelope:

```
let instance = person.extractInstancePerson(source);
...
xdmp.documentInsert(
  uri, person.instanceToEnvelope(instance, "xml"),
  ...)
```

Inside `person.instanceToEnvelope`, the `person.instanceToCanonical` function is called to create the `Person` entity embedded inside `es:envelope/es:instance`.

The table below illustrates the progression from raw data to XML envelope document, through use of the instance converter module functions.

Operation	Result

ingest raw source	{
	"pid": 2345,
	"given": "Martha",
	"family": "Washington"
	}

extractInstancePerson(source) input: raw source output: a map:map (json:object), shown here serialized as JSON	{"$attachments": {\"pid\":2345, \"given\":\"Martha\", \"family\":\"Washington\"}", "$type": "Person", "id": "2345", "firstName": "Martha", "lastName": "Washington", "fullName": "Martha Washington" }

Operation	Result

`instanceToCanonical(instance, "xml")`
 	
input: instance map:map
output: XML elem
	`<Person>`
	`  <id>2345</id>`
	`  <firstName>Martha</firstName>`
	`  <lastName>Washington</lastName>`
	`  <fullName>Martha Washington</fullName>`
	`</Person>`

`instanceToEnvelope(instance, "xml")`
 	
input: instance map:map
output: XML envelope doc
	`<es:envelope`
	`  xmlns:es="http://marklogic.com/entity-services">`
	`  <es:instance>`
	`  <es:info>`
	`  <es:title>Person</es:title>`
	`  <es:version>1.0.0</es:version>`
	`  </es:info>`
	`  <Person>`
	`  <id>2345</id>`
	`  <firstName>Martha</firstName>`
	`  <lastName>Washington</lastName>`
	`  <fullName>Martha Washington</fullName>`
	`  </Person>`
	`  </es:instance>`
	`  <es:attachments>{"pid":2345, "given":"Martha", "family":"Washington"}</es:attachments>`
	`</es:envelope>`

The following is an equivalent JSON envelope, generated by calling `instanceToEnvelope(instance, "json")`:

```
{ "envelope": {
  "instance": {
    "info": {
      "title":"Person",
      "version":"1.0.0"
    },
    "Person": {
      "id":"2345",
      "firstName":"Martha",
      "lastName":"Washington",
      "fullName":"Martha Washington"}
    },
    "attachments":[

"<person><pid>2345</pid><given>Martha</given><family>Washington</family></person>"
    ]
}}
```

Note that the source data in the attachments is as a string if it does not match the envelope data format. For example, in the above JSON envelope, the source attachment is a string, rather than an XML node. This has implications for extracting the source from the envelope as a node; see the example in “Query the Data” on page 51.

2.4.6 Query the Data

This section illustrates one way to search your entity instance data, using the JSearch API. You can also use other MarkLogic document search APIs, search your instances as row data, or use semantic search. The Entity Services API includes tools to facilitate all these forms of search. For details, see “Querying a Model or Entity Instances” on page 169.

The following example uses the JSearch API to find all `Person` entities with a `lastName` property of Washington.

1. Open Query Console in your browser if you do not already have it open.

2. Add a new query to the workspace by clicking on the “+” button on the query editor.

3. Select JavaScript in the Query Type dropdown.

4. Select your content database from the Database dropdown.

5. Copy and paste the following code into the new query. The code matches documents in the `person-envelopes` collection where the `es:instance` element includes a `lastName` element with the value “washington”, and then returns the original source data from the envelope.

```
'use strict';
import jsearch from '/MarkLogic/jsearch.mjs';

// Find all occurences of lastName with the value 'washington'
contained
// in an es:instance element. Return just the documents in the results.
const people = jsearch.collections('person-envelopes');
const matches = people.documents()
  .where(cts.elementQuery(
      fn.QName('http://marklogic.com/entity-services', 'instance'),
      cts.elementValueQuery('lastName', 'washington')))
  .map(match => match.document)
  .result();

// Extract the raw source data from the search results,
// as XML or JSON nodes
const asNodes = [];
for (let match of matches.results) {
  let attachment = fn.head(match.xpath('//*:attachments/node()'));
  if (attachment instanceof Element) {
    // already an XML node
    asNodes.push(attachment);
  } else {
```

```
    // serialized JSON; deserialize to a JSON document node
    asNodes.push(fn.head(xdmp.unquote(attachment)));
  }
}
// Dump the results in Query Console. The conversion from array
// to Sequence is just used to finesse the way QC renders array
// items that are XML nodes. It is not functionally significant.
Sequence.from(asNodes);
```

6. Click the Run button. You should see results similar to the following:

```
{ "pid":2345,
  "given":"Martha",
  "family":"Washington" }

<person xmlns:es="http://marklogic.com/entity-services">
  <pid>1234</pid>
  <given>George</given>
  <family>Washington</family>
</person>
```

The search matches two envelope documents, one extracted from JSON source and one extracted from XML source.

The search is first constrained to documents in the `person-envelopes` collection. Then a container query (`cts.elementQuery`) further constrains matches to those contained in an `es:instance` element. Finally, a value query (`cts.elementValueQuery`) is used to find elements named `lastName` with the value ‘washington’.

```
const people = jsearch.collections('person-envelopes');
const matches = people.documents()
  .where(cts.elementQuery(
      fn.QName('http://marklogic.com/entity-services', 'instance'),
      cts.elementValueQuery('lastName', 'washington')))
  ...
```

The container query ensures the search will not find matches in any part of the envelope data except the instance. You could similarly search just the attachments, though you cannot effectively perform a structured search on raw JSON data this way because JSON source is stored in the XML envelope document as a serialized string.

The map feature of JSearch is used to just return the matched documents, eliminating the search metadata such as the URI, relevance score, and confidence. The mapper was used just to streamline the output; a mapper is not required by Entity Services or the JSearch API.

```
people.documents()
  .where(...)
  .map(match => match.document)
```

The search produces the following output, which we saved to the `matches` variable for subsequent processing.

```
{"results":[
  <es:envelope xmlns:es="http://marklogic.com/entity-services">
    <es:instance>
      <es:info>
        <es:title>Person</es:title>
        <es:version>1.0.0</es:version>
      </es:info>
      <Person>
        <id>2345</id>
        <firstName>Martha</firstName>
        <lastName>Washington</lastName>
        <fullName>Martha Washington</fullName>
      </Person>
    </es:instance>
    <es:attachments>{"pid":2345, "given":"Martha",
"family":"Washington"}</es:attachments>
  </es:envelope>
  <es:envelope xmlns:es="http://marklogic.com/entity-services">
    <es:instance>
      <es:info>
        <es:title>Person</es:title>
        <es:version>1.0.0</es:version>
      </es:info>
      <Person>
        <id>1234</id>
        <firstName>George</firstName>
        <lastName>Washington</lastName>
        <fullName>George Washington</fullName>
      </Person>
    </es:instance>
    <es:attachments>
      <person>
        <pid>1234</pid>
        <given>George</given>
        <family>Washington</family>
      </person>
    </es:attachments>
  </es:envelope>
  ],
  "estimate":2
}
```

Note that the example code can return the original XML source data directly out of the envelope document because the attachments contain an XML element node. However, the original JSON source data must be converted from a string to a JSON node using `xdmp:from-json-string`, if you want to work with it as structured data. This conversion is the purpose of the following section of code:

```
if (attachment instanceof Element) {
  // already an XML node
  asNodes.push(attachment);
} else {
  // serialized JSON; deserialize to a JSON document node
  asNodes.push(fn.head(xdmp.fromJsonString(attachment)));
}
```

(The accumulation of the attachments into the `asNodes` array and subsequent conversion of `asNodes` into a `Sequence` is just done to finesse the way Query Console displays results.)

For more details and examples, see “Querying a Model or Entity Instances” on page 169.

2.4.7 Query the Model

When you created a model in “Create a Model” on page 41, MarkLogic automatically generated semantic triples from the descriptor. These triples define the model. You can add more “facts” about the model in the form of additional triples. You can use SPARQL or the Optic API to query a model.

For example, you can use a SPARQL query to discover what entity types are defined by a model, what properties are required in an entity instance of a particular type, or the datatype of a particular entity type property. For more details, see “Querying a Model or Entity Instances” on page 169.

The following procedure uses a SPARQL query to generate a list of all the required properties of an instance of the `Person` entity type:

1. Open Query Console in your browser if you do not already have it open.

2. Add a new query to the workspace by clicking on the “+” button on the query editor.

3. Select SPARQL Query in the Query Type dropdown.

4. Select your content database from the Database dropdown.

5. Copy and paste the following code into the new query. This code retrieves the names of all required properties of a `Person` entity instance.

```
prefix es:<http://marklogic.com/entity-services#>
select ?ptitle
where {
  ?x a es:EntityType;
       es:title "Person";
       es:property ?property .
  ?property a es:RequiredProperty;
              es:title ?ptitle
}
```

6. Click the Run button. The query results are displayed as a table.

You should see results similar to the following:

```
ptitle
"lastName"
"fullName"
"firstName"
```

You can also use the SQL and Optic APIs to query your model and entities as rows if you install an Entity Services generated TDE template based on your model. For more details and examples, see "Querying a Model or Entity Instances" on page 169. To learn more about Semantics in MarkLogic Server, see the *Semantics Developer's Guide*.

2.5 Next Steps

The following topics can help deepen your understanding of the Entity Services API.

- Explore the end to end Entity Services examples on GitHub. For details, see "Exploring the Entity Services Open-Source Examples" on page 15.
- Learn more about defining model descriptors; see "Creating and Managing Models" on page 57.

  Model descriptors support several features not covered here, such as identifying a primary key and flagging properties for indexing to facilitate fast searches.

- Learn about generating additional code and configuration artifacts from your model using the Entity Services API; see "Generating Code and Other Artifacts" on page 107.

  For example, you can use Entity Services to generate Search and Client API query options and database configuration artifacts based on your model. You can also generate a Template Driven Extraction (TDE) template that enables row and semantic search of instances. For details, see "Generating a TDE Template" on page 122.

- Learn more about querying models and instance data; see "Querying a Model or Entity Instances" on page 169.
- Explore the open source MarkLogic Data Hub project on GitHub (http://github.com/marklogic/marklogic-data-hub). Version 2.0 and later use the Entity Services API to create a Data Hub application that enables quick and easy entity modeling and creation of entities from source data.

3.0 Creating and Managing Models

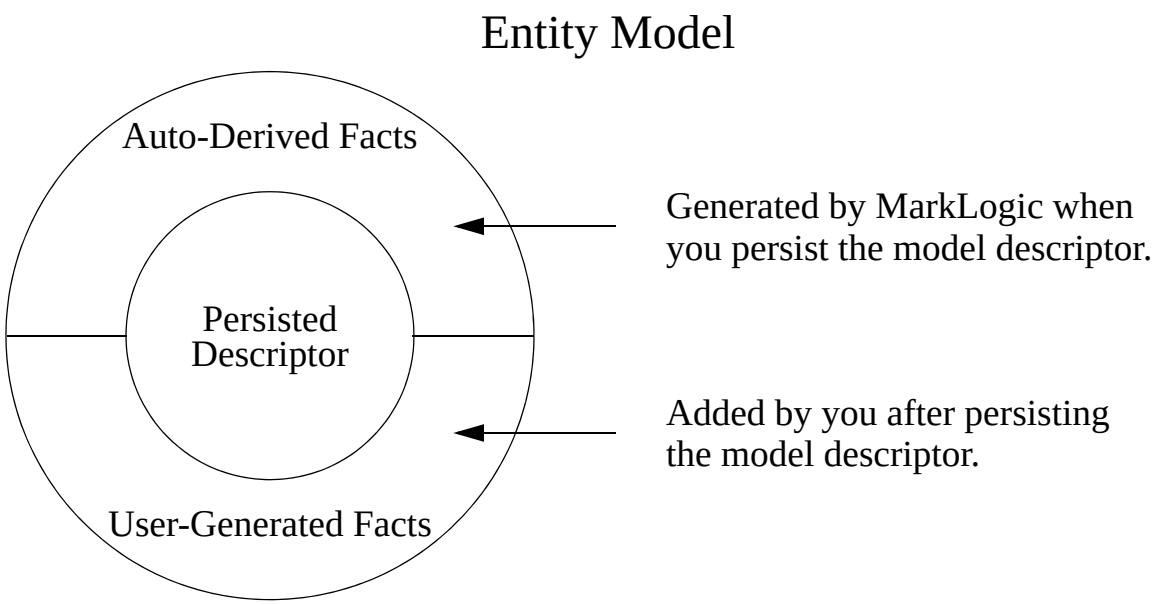
This chapter covers entity model description management tasks. A model descriptor defines entity types, their properties, and relationships between entities. The following topics are covered:

3.1 Introduction

A fully constructed model consists of a set of “facts” about the modeled entity types, their properties, and the relationships between them. The facts are represented in MarkLogic as semantic triples.

The entity types, properties, and relationships are defined by an XML or JSON model descriptor. When you persist the descriptor in the database in the prescribed way, MarkLogic automatically creates the model by generating facts about the model, expressed as triples. You can also add your own facts (triples) to the model.

The following diagram depicts the building blocks of an entity model in MarkLogic:



Building a model involves the following steps:

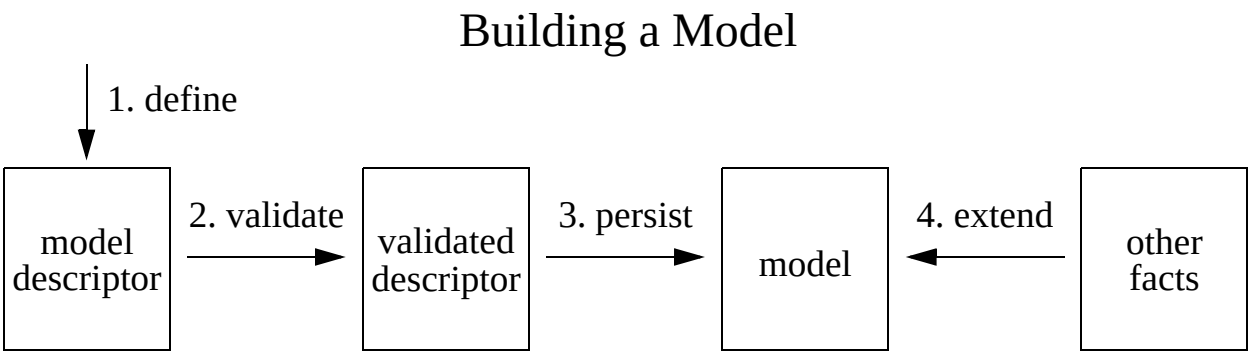
1. Define your entity types, entity type properties (attributes), and relationships in a model descriptor. For details, see “Writing a Model Descriptor” on page 59.

2. Optionally, validate your descriptor. An invalid descriptor will produce an invalid model, so it is a good idea to validate the descriptor during development. For details, see “Validating a Model Descriptor” on page 85.

3. Create a model by persisting the descriptor as a document in the special Entity Services collection. MarkLogic automatically generates facts about your entity types. For details, see “Creating a Model from a Model Descriptor” on page 83.

4. Optionally, extend the model with additional facts. “Extending a Model with Additional Facts” on page 87.

The following diagram is a pictoral representation of this process.



Once you have a valid descriptor or model, you can use the Entity Services API to generate code and other artifacts that provide a foundation for creating an application based on your model. You can use the API to create the following:

- A framework for transforming data from heterogeneous sources into canonical entity instances.
- A Template Driven Extraction (TDE) template for interfacing with your instance data as rows or triples. The template facilitates querying your instances using SQL, SPARQL, or the Optic API.
- A framework for converting instances from one version of your model to another as your model evolves and changes.
- Index configuration and other database configuration properties that facilitate querying your model, based on characteristics you define.
- Query options that facilitate full text search of your entity instances using the XQuery Search API or the REST, Java, and Node.js Client APIs.

For more details, see “Generating Code and Other Artifacts” on page 107.

3.2 Writing a Model Descriptor

This section describes how to define a model descriptor containing entity type definitions and model metadata. This section includes the following topics:

- Identifying the Primary Key Entity Property
- Identifying Personally Identifiable Information (PII)
- Distinguishing Required and Optional Entity Properties
- Defining a Namespace URI for an Entity Type
- Identifying Entity Properties for Indexing
- Controlling the Model IRI and Module Namespaces

3.2.1 Model Descriptor Basics

A model descriptor is an XML element or JSON object that defines one or more entity types, model metadata, and relationships between entity types. You can generate code templates and configuration artifacts from the descriptor in the form of either a JSON `object-node` or a `json:object` (a special kind of `map:map`).

A model descriptor has two parts: The `info` section contains model metadata, such as a title and version; the `definitions` section contains entity type definitions, including entity properties and relationships, plus type-specific metadata.

A descriptor must define at least one enity type and can define multiple entity types. Each type definition can include additional metadata to guide code and artifact generation. For details, see “Entity Type Definition Overview” on page 61.

> **Note:** The entity type property names in your model should be unique, even across entity types to avoid name collisions in generated code and artifacts.

The “natural” representation for a model descriptor is JSON because it already matches the internal representation of the model. When you use an XML model descriptor, you must call one of the following functions to translate your descriptor into a form usable with Entity Services functions that accept a model as input.

- XQuery: `es:model-from-xml` or `es:model-validate`
- Server-Side JavaScript: `es.modelFromXml` or `es.modelValidate`

For more details, see “Working With an XML Model Descriptor” on page 84.

You might find it useful to generate test entity instances during model development so you can see a concrete example of the default entities produced by your model. For details, see “Generating Test Entity Instances” on page 160.

The following is an example of the simplest possible model descriptor. The descriptor must contain at least `title` and `version` metadata in the `info` section, and define at least one entity type with at least one property in the `definitions` section. In this example, the model named "Example" defines an entity type named "Person". A `Person` entity has an `id` property.

Format	Descriptor Example

JSON	`{ "info": {`
	`  "title": "Example",`
	`  "version": "1.0.0"`
	`  },`
	`  "definitions": {`
	`  "Person": {`
	`  "properties": {`
	`  "id": { "datatype": "int" }`
	`  }`
	`}}}`

XML	`<es:model xmlns:es="http://marklogic.com/entity-services">`
	`  <es:info>`
	`  <es:title>Example</es:title>`
	`  <es:version>1.0.0</es:version>`
	`  </es:info>`
	`  <es:definitions>`
	`  <Person>`
	`  <es:properties>`
	`  <id>`
	`  <es:datatype>int</es:datatype>`
	`  </id>`
	`  </es:properties>`
	`  </Person>`
	`  </es:definitions>`
	`</es:model>`

3.2.2 Entity Type Definition Overview

An entity type definition usually includes one or more entity property definitions and can include the type metadata such as a primary key specification. This section provides a brief overview of defining an entity type. For syntax details, see "entity_type_definition" on page 96

All property definitions must include either a data type or a reference to another entity type. The data type of a property can be string, array, iri, or one of several XSD types. Depending on the data type, a property definition may require additional information. For details, see "Writing a Model Descriptor" on page 59 and "property_definition" on page 101.

The data type of an entity property can be any of the following:

- Any of the XSD types listed in "property_type" on page 105.

- A reference to another entity type.
- An IRI.
- A homogeneous array of items of any of these types.

Depending on the type, the property definition can include additional information. For example when the datatype is “string”, you can specify a collation. For syntax details, see “property_definition” on page 101.

An entity type definition can include the following type-specific metadata that is used when generating code and configuration artifacts:

- The name of an entity property to use as a primary key. Designation of a primary key affects semantic and row searches of instance data. For details, see “Identifying the Primary Key Entity Property” on page 67.
- Which entity properties must be present in every entity of this type. For details, see “Distinguishing Required and Optional Entity Properties” on page 70.
- Which entity properties should be backed by an index or lexicon. This affects database configuration and query option generation. For details, see “Identifying Entity Properties for Indexing” on page 75.
- A description of the entity type. This is purely informational and does not affect code or artifact generation.

Property names should be unique across all the entity types in a model. Duplicate property names can lead to name collisions in generated code and artifacts, causing some code and configuration to be generated commented out.

For example, the following model descriptor defines a Person entity with two required entity properties (“id” and “name”) and two optional entity properties (“address” and “rating”). The “id” property is the primary key. In addition, the descriptor specifies that a path range index configuration and query options should be generated for the “rating” property.

Language	Example

JSON	`{ "info": { "title": "Example", "version": "1.0.0" },`
	`  "definitions": {`
	`  "Person": {`
	`  "description": "Example person entity type",`
	`  "properties": {`
	`  "id": { "datatype": "int" },`
	`  "name": { "datatype": "string" },`
	`  "address": { "datatype": "string" },`
	`  "rating": { "datatype": "float" }`
	`  },`
	`  "required": ["id", "name"],`
	`  "primaryKey": "id",`
	`  "pathRangeIndex": ["rating"]`
	`  }`
	`  }`
	`}`

XML	`<es:model xmlns:es="http://marklogic.com/entity-services">`
	`  <es:info>`
	`  <es:title>Example</es:title>`
	`  <es:version>1.0.0</es:version>`
	`  </es:info>`
	`  <es:definitions>`
	`  <Person>`
	`  <es:description>Example person entity type</es:description>`
	`  <es:properties>`
	`  <id><es:datatype>int</es:datatype></id>`
	`  <name><es:datatype>string</es:datatype></name>`
	`  <address><es:datatype>string</es:datatype></address>`
	`  <rating><es:datatype>float</es:datatype></rating>`
	`  </es:properties>`
	`  <es:required>id</es:required>`
	`  <es:required>name</es:required>`
	`  <es:primary-key>id</es:primary-key>`
	`  <es:path-range-index>rating</es:path-range-index>`
	`  </Person>`
	`  </es:definitions>`
	`</es:model>`

3.2.3 Defining an Entity Property with a SimpleType

To define an entity type property with a simple type such as string, integer, or date, specify the type name as the value of the `datatype` JSON property or XML element. For a complete list of supported type names, see “property_type” on page 105.

Note: Not all the supported data types are usable as range index or word lexicon types. If you specify an entity property with an incompatible type in the range index or word lexicon specification of an entity type definition, then the resulting model will not validate.

For example, the following entity type definition contains entity properties with four different simple types.

Format	Example

JSON	`{ "info": { "title": "Example", "version": "1.0.0"},`
	`  "definitions": {`
	`  "Person": {`
	`  "properties": {`
	`  "id": { "datatype": "positiveInteger" },`
	`  "name": { "datatype": "string" },`
	`  "birthdate": { "datatype": "date" },`
	`  "rating": { "datatype": "float" }`
	`  }`
	`}}}`

XML	`<es:model xmlns:es="http://marklogic.com/entity-services">`
	`  <es:info>`
	`  <es:title>Example</es:title>`
	`  <es:version>1.0.0</es:version>`
	`  </es:info>`
	`  <es:definitions>`
	`  <Person>`
	`  <es:properties>`
	`  <id><es:datatype>positiveInteger</es:datatype></id>`
	`  <name><es:datatype>string</es:datatype></name>`
	`  <birthdate><es:datatype>date</es:datatype></birthdate>`
	`  <rating><es:datatype>float</es:datatype></rating>`
	`  </es:properties>`
	`  </Person>`
	`  </es:definitions>`
	`</es:model>`

If the type name is “string”, then you can optionally include a collation URI to be used when generating index, lexicon, and query option configuration artifacts from the model. If you omit the collation for a string-typed entity property, the collation defaults to “http://marklogic.com/collation/en”.

The following example demonstrates including a collation in an entity property definition.

Format	Example

JSON	`{ "info": { "title": "Example", "version": "1.0.0"},`
	`  "definitions": {`
	`  "Person": {`
	`  "properties": {`
	`  "name": {`
	`  "datatype": "string",`
	`  "collation": "http://marklogic.com/collation/"`
	`  }`
	`  }`
	`}}}`

XML	`<es:model xmlns:es="http://marklogic.com/entity-services">`
	`  <es:info>`
	`  <es:title>Example</es:title>`
	`  <es:version>1.0.0</es:version>`
	`  </es:info>`
	`  <es:definitions>`
	`  <Person>`
	`  <es:properties>`
	`  <name>`
	`  <es:datatype>string</es:datatype>`
	`  <es:collation>http://marklogic.com/collation/</es:collation>`
	`  </name>`
	`  </es:properties>`
	`  </Person>`
	`  </es:definitions>`
	`</es:model>`

3.2.4 Defining an Entity Property with a Complex Type

To specify an entity property whose type is complex, such as an object type, define the complex type as an entity type and use an entity type reference in the property definition.

For example, suppose a `Person` entity type contains a “name” property, and that “name” should have entity properties “first”, “middle”, and “last”. You could model a name as an entity type and then reference it in the definition of `Person` similar to the following:

```
JSON: "name": { "$ref": "#/definitions/Name" }

XML: <name><es:ref>#/definitions/Name</es:ref></name>
```

You can reference entity types defined in the same model (a local reference) or externally. For more details and examples, see “Defining Entity Relationships” on page 80.

3.2.5 Defining an Entity Property with Array Type

To specify an entity property whose type is a list of values, specify "array" in the `datatype` JSON property or XML element of the property definition, and then include an `items` type definition that specifies the data type of the list items. For a list of supported item type names, see "property_type" on page 105.

> **Note:** You cannot use an entity property with array type as a primary key or for generating database configuration artifacts such as range index or word lexicon configuration.

For example, the following entity type definition defines an entity property named "orders" whose value is an array of values of type "integer".

Format	Example

JSON	`{ "info": { "title": "Example", "version": "1.0.0"},`
	`  "definitions": {`
	`  "Person": {`
	`  "properties": {`
	`  "orders": {`
	`  "datatype": "array",`
	`  "items": {`
	`  "datatype": "integer"`
	`  }`
	`  }`
	`  }`
	`}}}`

XML	`<es:model xmlns:es="http://marklogic.com/entity-services">`
	`  <es:info>`
	`  <es:title>Example</es:title>`
	`  <es:version>1.0.0</es:version>`
	`  </es:info>`
	`  <es:definitions>`
	`  <Person>`
	`  <es:properties>`
	`  <orders>`
	`  <es:datatype>array</es:datatype>`
	`  <es:items>`
	`  <es:datatype>integer</es:datatype>`
	`  </es:items>`
	`  </orders>`
	`  </es:properties>`
	`  </Person>`
	`  </es:definitions>`
	`</es:model>`

For more details, see "property_definition" on page 101.

3.2.6 Defining an IRI Entity Property

To model the type of an entity property as an IRI (Internationalized Resource Identifier), specify "iri" in the `datatype` JSON property or XML element of the property definition. IRI-typed entity properties can be useful for working with entities using SPARQL.

The value of a property with IRI type must be a string that represents a `sem:iri` value. The value is opaque to the Entity Services API.

For example, the following entity type definition contains an entity property "name" with IRI data type.

Format	Example

JSON	`{ "info": { "title": "Example", "version": "1.0.0"},`
	`  "definitions": {`
	`  "Person": {`
	`  "properties": {`
	`  "name": { "datatype": "iri" }`
	`  }`
	`}}}`

XML	`<es:model xmlns:es="http://marklogic.com/entity-services">`
	`  <es:info>`
	`  <es:title>Example</es:title>`
	`  <es:version>1.0.0</es:version>`
	`  </es:info>`
	`  <es:definitions>`
	`  <Person>`
	`  <es:properties>`
	`  <name><es:datatype>iri</es:datatype></name>`
	`  </es:properties>`
	`  </Person>`
	`  </es:definitions>`
	`</es:model>`

For more details about creating Semantic applications in MarkLogic, see the *Semantics Developer's Guide*.

3.2.7 Identifying the Primary Key Entity Property

An entity type definition can designate one entity property as a primary key that uniquely identifies each instance of that type.

The primary key is used in the following ways:

- Primary and foreign key for SQL views of your instance data. If you generate a TDE template from the model, the primary key property is the primary key for a row view of

instance data. It is also used as a foreign key for some supporting views. For details, see “Generating a TDE Template” on page 122.

- Unique identifier for auto-generated instance facts (triples). If you generate a TDE template from the model, the template enables generation of triples about each instance of an entity type that defines a primary key. For details, see “Generating a TDE Template” on page 122.
- Value constraint on the primary key. If you generate query options from the model, the options pre-define a value constraint on the primary key. For details, see “Generating Query Options for Searching Instances” on page 141.

An entity type definition can contain at most one primary key. If you generate a schema from the model, the primary key entity property has its cardinality set to exactly 1; for details, see “Generating an Entity Instance Schema” on page 129.

To specify a primary key, include a `primaryKey` JSON property or `primary-key` XML element in the entity type definition. The value must be the name of an entity property defined in this type definition. The primary key entity property cannot have array type.

For example, the following definition of a Person entity defines the “id” entity property as a primary key:

Format	Example

JSON	`{ "info": { "title": "Example", "version": "1.0.0"},`
	`  "definitions": {`
	`  "Person": {`
	`  "properties": {`
	`  "id": { "datatype": "positiveInteger" },`
	`  "name": { "datatype": "string" }`
	`  },`
	`  **"primaryKey": "id"**`
	`}}}`

Format	Example
XML	<es:model xmlns:es="http://marklogic.com/entity-services"> <es:info> <es:title>Example</es:title> <es:version>1.0.0</es:version> </es:info> <es:definitions> <Person> <es:properties> <id><es:datatype>positiveInteger</es:datatype></id> <name><es:datatype>string</es:datatype></name> </es:properties> </Person> </es:definitions> **<es:primary-key>id</es:primary-key>** </es:model>

3.2.8 Identifying Personally Identifiable Information (PII)

Security policies often require strict access controls for Personally Identifiable Information (PII), such as a telephone number, address, or social security number. Entity Services enables you to tag entity properties as containing PII, and subsequently generate special security configuration to control access to PII data in your entity instances. For more details, see "Generating a PII Security Configuration Artifact" on page 134.

The following example entity type definition flags the "name" and "address" entity properties as PII.

Format	Example

JSON	{ "info": { "title": "Example", "version": "1.0.0"},
	  "definitions": {
	  "Person": {
	  "description": "Example person entity type",
	  "properties": {
	  "id": { "datatype": "int" },
	  "name": { "datatype": "string" },
	  "address": { "datatype": "string" }
	  },
	  "pii" : ["name", "address"],
	  "required": ["id", "name"]
	  }
	  }
	}

Format	Example
XML	<es:model xmlns:es="http://marklogic.com/entity-services"> <es:info> <es:title>Example</es:title> <es:version>1.0.0</es:version> </es:info> <es:definitions> <Person> <es:description>Example person entity type</es:description> <es:properties> <id><es:datatype>int</es:datatype></id> <name><es:datatype>string</es:datatype></name> <address><es:datatype>string</es:datatype></address> </es:properties> <es:pii>name</es:pii> <es:pii>address</es:pii> <es:required>id</es:required> <es:required>name</es:required> </Person> </es:definitions> </es:model>

3.2.9 Distinguishing Required and Optional Entity Properties

By default, all entity properties defined in an entity type are optional. You can identify required properties by including their names in the `required` section of your entity type definition. The entity properties named in the `required` section must be defined in the containing entity type.

An entity property specified as a primary key is implicitly required, so you should not also include it in the explicit list of required properties.

When you validate an instance against the schema generated for an instance type, validation fails if the instance does not include at least one occurrence of a required entity property. Similarly, when you generate a TDE template for an instance type, required entity properties are not considered nullable.

The following example entity type definition defines 3 entity properties: “id”, “name”, and “address”. The “id” and “name” properties are required. The “address” entity property is optional.

Format	Example

JSON	`{ "info": { "title": "Example", "version": "1.0.0"},`
	`  "definitions": {`
	`  "Person": {`
	`  "description": "Example person entity type",`
	`  "properties": {`
	`  "id": { "datatype": "int" },`
	`  "name": { "datatype": "string" },`
	`  "address": { "datatype": "string" }`
	`  },`
	`  **"required": ["id", "name"]**`
	`  }`
	`  }`
	`}`

XML	`<es:model xmlns:es="http://marklogic.com/entity-services">`
	`  <es:info>`
	`  <es:title>Example</es:title>`
	`  <es:version>1.0.0</es:version>`
	`  </es:info>`
	`  <es:definitions>`
	`  <Person>`
	`  <es:description>Example person entity type</es:description>`
	`  <es:properties>`
	`  <id><es:datatype>int</es:datatype></id>`
	`  <name><es:datatype>string</es:datatype></name>`
	`  <address><es:datatype>string</es:datatype></address>`
	`  </es:properties>`
	`  **<es:required>id</es:required>**`
	`  **<es:required>name</es:required>**`
	`  </Person>`
	`  </es:definitions>`
	`</es:model>`

3.2.10 Defining a Namespace URI for an Entity Type

By default, the elements of an XML entity instance are in no namespace. If you include a namespace URI and prefix in your model, then your entity instances names will be qualified by the namespace, as long as you use an XML representation for your envelope documents.

Use of entity type namespaces is optional. If you choose to use a namespace, you must specify both a namespace URI and a prefix in your entity type definition.

In an XML model descriptor, use the following format to define a namespace URI and prefix:

```
<es:namespace>namespaceURI</es:namespace>
<es:namespace-prefix>prefix</es:namespace-prefix>
```

In a JSON model descriptor, use the following format to define a namespace URI and prefix:

```
"namespace": "namespaceURI",
"namespacePrefix": "prefix"
```

The following restrictions apply to defining namespace prefix binding. Any model that violates these restrictions will fail validation.

- No namespace prefix can begin with “xml”, in any case combination. See https://www.w3.org/TR/REC-xml-names/.
- The following namespace prefixes are reserved and must not be used: xsi, xs, xsd, es, json. In general, you should not use namespace prefixes pre-defined by MarkLogic, such “xdmp”.
- The namespace XML element or JSON property value must be a valid absolute URI.
- Entity type namespace prefixes must be unique across the model. You cannot define multiple entity types with the same namespace prefix.

If you define a namespace for an entity type, the Entity Services API uses it when creating XML envelope documents, extracting instances from XML envelopes, and generating model artifacts such as schemas, query options, and TDE templates.

Note: The namespace is discarded when generating JSON envelope documents or extracting an instance from an envelope document as JSON. This means that generated code, query options, and TDE templates based on the model will include XPath expressions that will not match your JSON envelopes or instances without modification.

For example, the following model descriptor specifies that Person entities should be in the namespace “http://example.org/es/gs” and bind that namespace URI to the prefix “esgs”:

```
<es:model xmlns:es="http://marklogic.com/entity-services">
  <es:info>
    <es:title>Person</es:title>
    <es:version>1.1.0</es:version>
    <es:base-uri>http://example.org/example-person/</es:base-uri>
  </es:info>
  <es:definitions>
    <Person>
      <es:properties>
        <id><es:datatype>string</es:datatype></id>
        <firstName><es:datatype>string</es:datatype></firstName>
        <lastName><es:datatype>string</es:datatype></lastName>
        <fullName><es:datatype>string</es:datatype></fullName>
```

```
            <friends>
              <es:datatype>array</es:datatype>
              <es:items><es:ref>#/definitions/Person</es:ref></es:items>
            </friends>
          </es:properties>
          <es:namespace>http://example.org/es/gs</es:namespace>
          <es:namespace-prefix>esgs</es:namespace-prefix>
          <es:primary-key>id</es:primary-key>
          <es:required>firstName</es:required>
          <es:required>lastName</es:required>
          <es:required>fullName</es:required>
        </Person>
      </es:definitions>
    </es:model>
```

The following table illustrates how the envelope documents change, based on whether or not the model defines an entity type namespace.

Use Case	Example Envelope

No namespace in Person entity type definition	`<es:envelope`
	`  xmlns:es="http://marklogic.com/entity-services">`
	`  <es:instance>`
	`  <es:info>`
	`  ...`
	`  </es:info>`
	`  **<Person>**`
	`  **<id>1234</id>**`
	`  **<firstName>George</firstName>**`
	`  **<lastName>Washington</lastName>**`
	`  **<fullName>George Washington</fullName>**`
	`  **</Person>**`
	`  </es:instance>`
	`  <es:attachments>`
	`  ...`
	`  </es:attachments>`
	`</es:envelope>`

Person entity type definition defines namespace URI "http://example.org/es/gs" with prefix "esgs"	`<es:envelope`
	`  xmlns:es="http://marklogic.com/entity-services">`
	`  <es:instance>`
	`  <es:info>`
	`  ...`
	`  </es:info>`
	`  **<esgs:Person**`
	`  **xmlns:esgs="http://example.org/es/gs">**`
	`  **<esgs:id>1234</esgs:id>**`
	`  **<esgs:firstName>George</esgs:firstName>**`
	`  **<esgs:lastName>Washington</esgs:lastName>**`
	`  **<esgs:fullName>George Washington</esgs:fullName>**`
	`  **</esgs:Person>**`
	`  </es:instance>`
	`  <es:attachments>`
	`  ...`
	`  </es:attachments>`
	`</es:envelope>`

If you call `es:instance-xml-from-document` or `es.instanceXmlFromDocument` on an XML envelope document for an entity type that uses namespaces, the returned instance includes the namespace.

For example, the following instance is extracted from the envelope document shown in the table above. Notice that it uses the “esgs” namespace.

```
<esgs:Person xmlns:es="http://marklogic.com/entity-services"
             xmlns:esgs="http://example.org/es/gs">
  <esgs:id>1234</esgs:id>
```

```
    <esgs:firstName>George</esgs:firstName>
    <esgs:lastName>Washington</esgs:lastName>
    <esgs:fullName>George Washington</esgs:fullName>
</esgs:Person>
```

The namespace is not preserved when you use JSON envelopes or when you generate a JSON instance from an XML or JSON envelope.

3.2.11 Identifying Entity Properties for Indexing

Searchable entity properties should usually be backed by an index or lexicon.

A model descriptor can contain optional range index and word lexicon sections that indicate which entity properties should have an associated range index or word lexicon and search constraint definition. This specification affects generated artifacts such as query options and database configuration.

For more details, see the following topics:

- Specifying Indexable Properties
- Interaction with Generated Artifacts
- Example: Identifying Indexable Entity Properties
- Supported Datatypes

3.2.11.1 Specifying Indexable Properties

A range index enables range queries over an entity property, such as “match all inventory item instances with a price property greater than 5”. Range indexes and word lexicons also enable search application features such as faceting and search term suggestions.

The Entity Services modeling language enables you to specify entity type properties that should be backed by an element range index, path range index, or word lexicon. (The element range index specification is applicable to both XML elements and JSON properties.)

To indicate that a property should be backed by a range index, include the following components in your model descriptor:

- JSON: `pathRangeIndex` or `elementRangeIndex`
- XML: `es:path-range-index` or `es:element-range-index`

In JSON, the value of `pathRangeIndex` and `elementRangeIndex` is an array of entity property names. In XML, define multiple `path-range-index` or `element-range-index` elements to tag multiple properties. For example:

```
JSON: "pathRangeIndex": ["price", "rating"]
```

```
XML: <es:path-range-index>price</es:path-range-index>
     <es:path-range-index>rating</es:path-range-index>
```

Note that an element range index is applicable to both XML elements and JSON properties, so your choice of index type is not limited by the representation of your entity instances. For details, see Creating Indexes and Lexicons Over JSON Documents in the *Application Developer's Guide*.

To specify properties to be backed by a word lexicon, include a `wordLexicon` JSON property or `word-lexicon` XML element in your model descriptor. In JSON, the value of `wordLexicon` is an array of property names. In XML, define multiple `word-lexicon` elements to tag multiple properties. The syntax is analogous to the range index example, above.

The properties named in a range index or word lexicon specification must be defined in the containing entity type definition and must conform to certain data type restrictions. For data type details, see "Supported Datatypes" on page 78.

For a complete example, see "Example: Identifying Indexable Entity Properties" on page 77.

3.2.11.2 Interaction with Generated Artifacts

Specifying the name of an entity property in the range index section has the following implications:

- The database properties generated by the `es:database-properties-generate` XQuery function or the `es.databasePropertiesGenerate` JavaScript function will include path range index configuration for the named entity property.
- The query options generated by the `es:search-options-generate` XQuery function or the `es.searchOptionsGenerate` JavaScript function will include a path range constraint definition for the named entity property.

Specifying the name of an entity property in the word lexicon section has the following implications:

- The database properties generated by the `es:database-properties-generate` XQuery function or the `es.databasePropertiesGenerate` JavaScript function will include word lexicon configuration for the named entity property.
- The query options generated by the `es:search-options-generate` XQuery function or the `es.searchOptionsGenerate` JavaScript function will include a word constraint definition for the named entity property.

  Note: If your model specifies a namespace binding for an entity type and you use JSON envelopes, the namespace is discarded in the JSON representation, but the generated index configuration still assumes a namespace, so the index configuration will not match your JSON data. You should usually use XML envelopes when you include a namespace specifier in your model.

For more details, see “Generating a Database Configuration Artifact” on page 137 and “Generating Query Options for Searching Instances” on page 141.

3.2.11.3 Example: Identifying Indexable Entity Properties

The following example descriptors specify a path range index on the “rating” entity property and a word lexicon on the “bio” entity property of a “Person” entity type.

Format	Model Descriptor Example

JSON	`{ "info": { "title": "Example", "version": "1.0.0"},`
	`  "definitions": {`
	`  "Person": {`
	`  "description": "Example person entity type",`
	`  "properties": {`
	`  "id": { "datatype": "int" },`
	`  "name": { "datatype": "string" },`
	`  "rating": { "datatype": "float" },`
	`  "bio": { "datatype": "string" }`
	`  },`
	`  **"pathRangeIndex": ["rating"],**`
	`  **"wordLexicon": ["bio"]**`
	`  }`
	`  }`
	`}`

XML	`<es:model xmlns:es="http://marklogic.com/entity-services">`
	`  <es:info>`
	`  <es:title>Example</es:title>`
	`  <es:version>1.0.0</es:version>`
	`  </es:info>`
	`  <es:definitions>`
	`  <Person>`
	`  <es:description>Example person entity type</es:description>`
	`  <es:properties>`
	`  <id><es:datatype>int</es:datatype></id>`
	`  <name><es:datatype>string</es:datatype></name>`
	`  <rating><es:datatype>float</es:datatype></rating>`
	`  <bio><es:datatype>string</es:datatype></bio>`
	`  </es:properties>`
	`  **<es:path-range-index>rating</es:path-range-index>**`
	`  **<es:word-lexicon>bio</es:word-lexicon>**`
	`  </Person>`
	`  </es:definitions>`
	`</es:model>`

If you generate database properties from the resulting model (using `es:database-properties-generate` or `es.databasePropertiesGenerate`), then the generated database configuration properties include the following details:

```
{ "database-name":"%%DATABASE%%",
  ...,
  "element-word-lexicon":[{
    "collation":"http://marklogic.com/collation/en",
    "localname":"bio",
    "namespace-uri":""
  }],
  "range-path-index":[{
    "collation":"http://marklogic.com/collation/en",
    "invalid-values":"reject",
    "path-expression":"//es:instance/Person/rating",
    "range-value-positions":false,
    "scalar-type":"float"
  }],
  ...
}
```

If you generate query options from the resulting model (using `es:search-options-generate` or `es.searchOptionsGenerate`), then the generated options include the following constraint definitions:

```
<search:options
    xmlns:search="http://marklogic.com/appservices/search">
  ...
  <search:constraint name="rating">
    <search:range type="xs:float" facet="true">
      <search:path-index xmlns:es=...>
        //es:instance/Person/rating
      </search:path-index>
    </search:range>
  </search:constraint>
  <search:constraint name="bio">
    <search:word>
      <search:element ns="" name="bio"/>
    </search:word>
  </search:constraint>
  ...
</search:options>
```

For details on generating database properties and query options, see “Generating Code and Other Artifacts” on page 107. For details on using the generated artifacts, see “Deploying Generated Code and Artifacts” on page 147 and “Querying a Model or Entity Instances” on page 169.

3.2.11.4 Supported Datatypes

Any property named in a range index specification must have a data type that can be used to define a range index or can be mapped to an indexable super type. You can define a property with any of the data types listed in “property_type” on page 105, but only scalar types can be used to define a range index. For example, you cannot specify a property that has type `hexBinary`, an array type, or a reference to another entity type.

For a list of type usable to define range indexes, see Understanding Range Indexes in the *Administrator's Guide*.

Any entity property specified in the word lexicon section of a model descriptor must have string type, or a type which normalizes to string, such as `anyURI` or `iri`.

Some datatypes are normalized to a supported index type for purposes of index configuration. For example, the `positiveInteger`, `negativeInteger`, and `integer` datatypes normalize to the XSD `decimal` type. The following mapping is used for purposes of index configuration:

- `byte`, `short` become `int`
- `unsignedByte`, `unsignedShort` become `unsignedInt`
- all `*integer` types become `decimal`
- `iri`, `anyURI`, `boolean` become `string`

3.2.12 Controlling the Model IRI and Module Namespaces

The info section of a model descriptor can include an optional `base-uri` XML element or `baseUri` JSON property. If a base URI is defined, it is used for the following purposes:

- When you use Entity Services to generate code modules such as an instance converter, the module namespace uses the base URI as the beginning of the module namespace URI.
- When you generate a model from the descriptor, the base URI is used as the beginning of the model IRI when generating facts about the model as RDF triples.

If you do not include a base URI definition in your descriptor, Entity Services uses "http://example.org/".

For example, the following descriptor defines a base URI of "http://my/org/".

Format	Model Descriptor Example

JSON	`{ "info": {`
	`  "title": "Example",`
	`  "version": "1.0.0"`
	`  **"baseUri": "http://my/org/"`**
	`  },`
	`  "definitions": {`
	`  "Person": { ... }`
	`  }`
	`}`

Format	Model Descriptor Example
XML	<es:model xmlns:es="http://marklogic.com/entity-services"> <es:info> <es:title>Example</es:title> <es:version>1.0.0</es:version> **<es:base-uri>http://my/org/</es:base-uri>** </es:info> <es:definitions> <Person>...</Person> </es:definitions> </es:model>

If you generate an instance converter module from this descriptor, then the module namespace is created by appending the module title (Example) and version (1.0.0) to the base URI ("http://my/org/"), as follows:

```
module namespace example = "http://my/org/Example-1.0.0";
```

If you did not define a base URI, then the module namespace URI would be "http://example.org/Example-1.0.0". For more details on the generated module namespace, see "Module Namespace Declaration" on page 112.

Similarly, when you create a model from the above example descriptor, the base URI is used as an IRI prefix for the generated model and instance triples. For example, the Person entity type defined by the example has the following IRI:

```
http://my/org/Example-1.0.0/Person
```

If you do not define a base URI, then the above IRI would be `http://example.org/Example-1.0.0/Person`.

The base URI is always combined with other model metadata, such as the model title and version.

3.3 Defining Entity Relationships

You can model relationships between entity types by referencing an entity type in place of a `datatype` in the definition of an entity property. This is the `$ref` JSON property or `es:ref` XML element of the property definition.

References can either be local (identifying a type defined in the same descriptor) or external (identifying a type that cannot be locally resolved by the Entity Services API).

- Defining a Local Entity Reference
- Defining an External Entity Reference

3.3.1 Defining a Local Entity Reference

A local entity reference refers to an entity type defined in the current model. A local reference is defined by a relative URI of the following form:

```
#/definitions/entityTypeName
```

A local entity reference is resolvable during code generation, such as when you call the `es:instance-converter-generate` XQuery function or the `es.instanceConverterGenerate` JavaScript function. This resolvability enables the Entity Services code generation tools to, for example, embed the properties of a local reference inside an instance of the referencing type.

For example, the following model descriptor defines two entity types, “Person” and “Name”. The “Person” entity type definition includes a “name” entity property that is a reference to the “Name” entity type. The type of the “name” property is a local reference.

Format	Example

JSON	`{ "info": { "title": "Example", "version": "1.0.0"},`
	`  "definitions": {`
	`  "Name": {`
	`  "description": "The name of a person.",`
	`  "properties": {`
	`  "first": { "datatype": "string" },`
	`  "middle": { "datatype": "string" },`
	`  "last": { "datatype": "string" }`
	`  },`
	`  "required": ["first", "last"]`
	`  },`
	`  "Person": {`
	`  "description": "Example person entity type",`
	`  "properties": {`
	`  "id": { "datatype": "int" },`
	`  **"name": { "$ref": "#/definitions/Name"** },`
	`  }`
	`}}}`

Format	Example
XML	<es:model xmlns:es="http://marklogic.com/entity-services"> <es:info> <es:title>Example</es:title> <es:version>1.0.0</es:version> </es:info> <es:definitions> <Name> <es:description>The name of a person.</es:description> <es:properties> <first><es:datatype>string</es:datatype></first> <middle><es:datatype>string</es:datatype></middle> <last><es:datatype>string</es:datatype></last> </es:properties> <es:required>first</es:required> <es:required>last</es:required> </Name> <Person> <es:description>Example person entity type</es:description> <es:properties> <id><es:datatype>int</es:datatype></id> **<name><es:ref>#/definitions/Name</es:ref></name>** </es:properties> </Person> </es:definitions> </es:model>

If you generate an instance converter from this model, the default code template assumes that a Person entity instance has a Name entity instance embedded within it. For example, a Person entity instance generated by `es:instance-json-from-document` or `es.instanceJsonFromDocument` might look like the following:

```
{ "Person": {
    "id": 1234,
    "name": {
      "first": "John",
      "middle": "NMI",
      "last": "Smith"
    }
} }
```

You could also choose to have the Name persisted separately and reference it from a Person entity via a primary key, URI, or other identifier. That is a choice you make when customizing your instance converter. For more details, see “Creating an Instance Converter Module” on page 110.

3.3.2 Defining an External Entity Reference

An external entity reference refers to an entity type defined outside the model. The referenced type is identified by an IRI. The referenced type should be defined elsewhere in MarkLogic. Resolution of the reference is handled by MarkLogic’s SPARQL engine.

No validation is performed on the value of an external reference. When you use the Entity Services APIs to generate code and other artifacts, the reference is treated as an opaque string.

For example, the following model descriptor defines a “Person” entity type that contains a “name” property that is an external reference to a type identified by “http://example.org/Name”. This could be an entity type defined by a different Entity Services model.

Format	Example

JSON	`{ "info": { "title": "Example", "version": "1.0.0"},`
	`  "definitions": {`
	`  "Person": {`
	`  "description": "Example person entity type",`
	`  "properties": {`
	`  "id": { "datatype": "int" },`
	`  **"name": { "$ref": "http://example.org/Name"** },`
	`  }`
	`}}}`

XML	`<es:model xmlns:es="http://marklogic.com/entity-services">`
	`  <es:info>`
	`  <es:title>Example</es:title>`
	`  <es:version>1.0.0</es:version>`
	`  </es:info>`
	`  <es:definitions>`
	`  <Person>`
	`  <es:description>Example person entity type</es:description>`
	`  <es:properties>`
	`  <id><es:datatype>int</es:datatype></id>`
	`  **<name><es:ref>http://example.org/Name</es:ref></name>**`
	`  </es:properties>`
	`  </Person>`
	`  </es:definitions>`
	`</es:model>`

You would customize your `Person` instance converter code to fill in the value of the name property with an appropriate reference or embedded value. Since the “shape” of the external entity type is not defined by the model, the Entity Services code generation tools cannot assume an embedded object as they can for local references. To learn more about instance generation, see “Creating an Instance Converter Module” on page 110.

3.4 Creating a Model from a Model Descriptor

Create a model from a JSON or XML descriptor by inserting the descriptor document into the database as part of the special Entity Services collection `http://marklogic.com/entity-services/models`.

During insertion, MarkLogic generates a model from the descriptor. The model includes the entity type definitions, properties, and relationships defined by your descriptor, plus facts about the model that MarkLogic automatically infers from the descriptor. These facts are expressed as Semantic triples; for details, see "Search Basics for Models" on page 170. You can also add your own facts; for details, see "Extending a Model with Additional Facts" on page 87.

For example, the following code snippet creates a model from a descriptor. For a more complete example see "Getting Started With Entity Services" on page 19.

Language	Example

XQuery	`xquery version "1.0-ml";`
	`import module namespace es = "http://marklogic.com/entity-services"`
	`  at "/MarkLogic/entity-services/entity-services.xqy";`
	
	`let $desc := ... (: your model descriptor :)`
	`return xdmp:document-insert(`
	`  '/es-gs/models/person-1.0.0.json', $desc,`
	`  <options xmlns="xdmp:document-insert">`
	`  <collections>{`
	`  <collection>http://marklogic.com/entity-services/models</collection>,`
	`  for $coll in xdmp:default-collections()`
	`  return <collection>{$coll}</collection>`
	`  }</collections>`
	`  </options>`
	`)`

JavaScript	`'use strict';`
	`declareUpdate();`
	`const es = require('/MarkLogic/entity-services/entity-services.xqy');`
	
	`const desc = ...; // your model descriptor`
	`xdmp.documentInsert(`
	`  '/es-gs/models/person-1.0.0.json', desc,`
	`  {collections: ['http://marklogic.com/entity-services/models']}`
	`);`

Note that if you create a model with an XML descriptor, then you will have to convert the persisted document to its in-memory JSON representation before you can use it with any Entity Services functions that expect a model as input. For details, see "Working With an XML Model Descriptor" on page 84.

3.5 Working With an XML Model Descriptor

The "natural" representation of a model descriptor in the Entity Services API is a JSON object node. In XQuery, the in-memory JSON representation of a model descriptor is as a `json:object` (a special kind of map:map). The equivalent representation in Server-Side JavaScript is a JSON object node or JavaScript object. (MarkLogic implictly converts JavaScript objects to JSON objects when you pass them as parameters.)

If you create a model by persisting an XML descriptor, you must convert the persisted descriptor into its JSON representation before you can pass it to most Entity Services functions. You can create a JSON object from an XML descriptor using the following functions:

- XQuery:`es:model-validate` or `es:model-from-xml`
- Server-Side JavaScript: `es.modelValidate` or `es.modelFromXml`

To learn more about descriptor validation, see "Validating a Model Descriptor" on page 85.

The following example code snippet generates an instance converter module from an XML descriptor by first converting the descriptor to JSON. Assume `/es-gs/models/person-1.0.0.xml` is previously persisted descriptor used to create a model.

Language	Example

XQuery	`xquery version "1.0-ml";`
	`import module namespace es = "http://marklogic.com/entity-services"`
	`  at "/MarkLogic/entity-services/entity-services.xqy";`
	
	`let $desc := fn:doc('/es-gs/models/person-1.0.0.xml')`
	`return es:instance-converter-generate(`
	`  es:model-from-xml($desc))`

JavaScript	`'use strict';`
	`const es = require('/MarkLogic/entity-services/entity-services.xqy');`
	
	`const desc = cts.doc('/es-gs/models/person-1.0.0.xml');`
	`es.instanceConverterGenerate(es.modelFromXml(desc));`

If you persist your XML descriptor as JSON instead of XML, then you only need to do the conversion once, at model creation time. This is the technique used in "Create a Model" on page 25.

In XQuery, you can manipulate the JSON representation of the descriptor as a `map:map`; for details, see "Building a JSON Object from a Map" on page 371.

3.6 Validating a Model Descriptor

To validate a model descriptor, use the `es:model-validate` XQuery function or the `es.modelValidate` Server-Side JavaScript function.

If the input descriptor is valid, this function returns a valid JSON descriptor that can be persisted in the database or used as input to any Entity Services interfaces that accepts a model as input. If the input descriptor is invalid, this function throws an `ES-MODEL-INVALID` exception and reports the validation failures in the error details.

Since an invalid model descriptor produces an invalid model, you should use model validation during development. Model validation does introduce added overhead, however, so you might choose to skip it when going between a descriptor and a model in production situations.

The following example validates a simple model descriptor containing a “Person” entity type definition. The model descriptor is valid, so no exception is raised, and the returned model is identical to the JSON model descriptor used in the JavaScript example.

Language	Example

XQuery	`xquery version "1.0-ml";`
	`import module namespace es =`
	`"http://marklogic.com/entity-services"`
	`  at "/MarkLogic/entity-services/entity-services.xqy";`
	`es:model-validate(`
	`<es:model xmlns:es="http://marklogic.com/entity-services">`
	`  <es:info>`
	`  <es:title>Example</es:title>`
	`  <es:version>1.0.0</es:version>`
	`  </es:info>`
	`  <es:definitions>`
	`  <Person>`
	`  <es:properties>`
	`  <id><es:datatype>int</es:datatype></id>`
	`  <name><es:datatype>string</es:datatype></name>`
	`  </es:properties>`
	`  <es:required>id</es:required>`
	`  <es:required>name</es:required>`
	`  </Person>`
	`  </es:definitions>`
	`</es:model>`
	`)`

JavaScript	`var es = require('/MarkLogic/entity-services/entity-services');`
	`es.modelValidate(`
	`  { "info": { "title": "Example", "version": "1.0.0" },`
	`  "definitions": {`
	`  "Person": {`
	`  "properties": {`
	`  "id": { "datatype": "int" },`
	`  "name": { "datatype": "string" },`
	`  },`
	`  "required": ["id", "name"]`
	`  } } }`
	`);`

If we introduce an error by specifying that an undefined entity property named “UNDEF” is a required property, then validation raises an error similar to the following:

```
ES-MODEL-INVALID (err:FOER0000): "Required" property UNDEF doesn't
exist.
```

3.7 Extending a Model with Additional Facts

You can extend your model with information and relationships that cannot be expressed in or derived from the model descriptor by storing additional semantic triples related to your model in MarkLogic.

You can use the model, entity type, and property IRIs generated by Entity Services to express these new facts. Entity Services uses the following patterns for constructing IRIs when generating RDF triple data about a model:

- model IRI: *baseUri/modelTitle-modelVersion*
- entity type IRI: *modelIri/typeName*
- entity property IRI: *entityTypeIri/propertyName*

For example, suppose you have the following model descriptor:

```
{ "info": {
    "title": "People",
    "version": "1.0.0",
    "baseUri": "http://marklogic.com/example/"
  },
  "definitions": {
    "Person": {
      "properties": {
        "id": { "datatype": "int" },
        "name": { "datatype": "string" },
      }
} } }
```

Then the following IRIs are generated and used by Entity Services:

- People model IRI: `http://marklogic.com/example/People-1.0.0`
- Person entity type IRI: `http://marklogic.com/example/People-1.0.0/Person`
- Person property “name” IRI: `http://marklogic.com/example/People-1.0.0/Person/name`

You can use any of MarkLogic’s Semantic capabilities to add, manage, and query triples you add to your model, including embedding triples in your entity instance envelope documents and customizing the TDE template you can generate with Entity Services. You can also use the model IRI as named graph IRI for integrating separate triples-based modeling with an Entity Services model.

For more information about using Semantics with MarkLogic, see *Semantics Developer’s Guide*.

3.8 Managing Model Changes

Some kinds of changes do not affect the structure and content of your instances. For example, if you decide to index a property that was not previously indexed or change a property from required to optional, your instances will not change.

However, changes such as the following typically impact the content in your instances, application code, and generated artifacts:

- add or remove a property
- change the data type of a property
- make an optional property required
- add or remove an entity type

Entity Services can help you update your application as your model evolves.

When integrating model changes, you must decide if all consumers of your instance data will move to the new model at the same time, or if you need to support both old and new models during some transition period. You must also choose how to generate instances based on your new model version.

See the following topics for more details:

- Generating Instances From the New Model
- Replacing the Old Model with a New Version
- Making Multiple Model Versions Available

For an end to end example of updating a model version, see `example-versions` in the Entity Services examples on GitHub. For more details, see “Exploring the Entity Services Open-Source Examples” on page 15.

3.8.1 Generating Instances From the New Model

You can upgrade your instance data using one of the following strategies:

- Re-extract instances from original source using an instance converter generated from the new model.
- Convert old version instances into new using a version translator.

What you do with the instance data based on the new model depends on your version transition strategy. For details, see “Replacing the Old Model with a New Version” on page 90 and “Making Multiple Model Versions Available” on page 90.

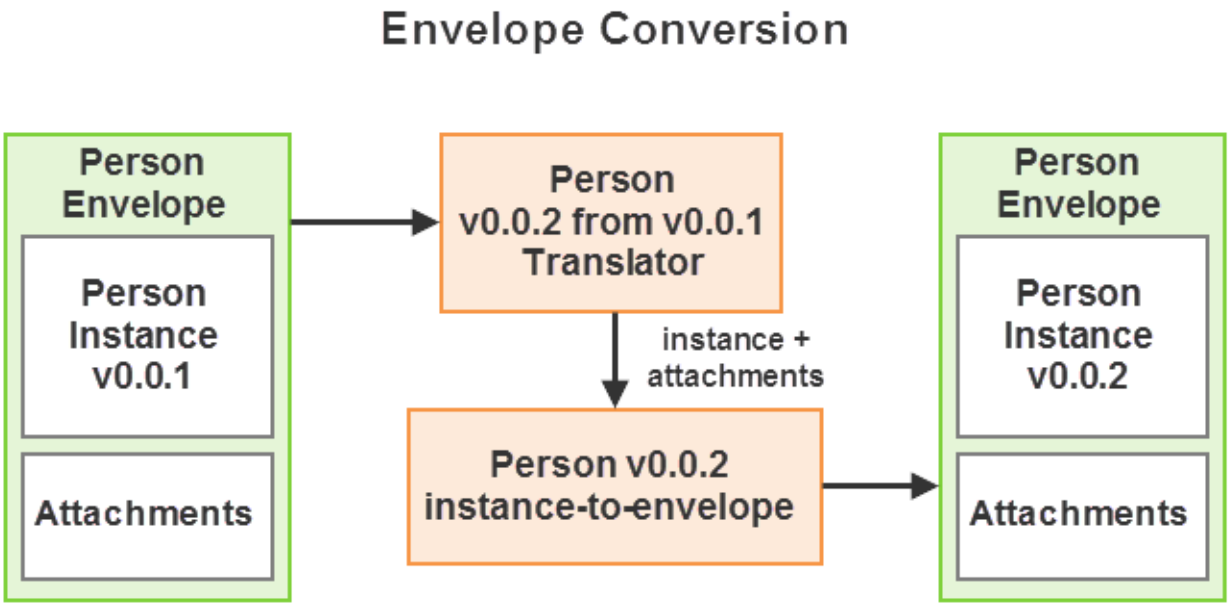
You should use a version translator if re-extraction is not practical. A version translator is also useful for creating in-memory instances of a different version to return to downstream consumers. For example, if you’ve advanced your content to v2 of your model, you could use a v2-to-v1 translator to synthesize v1 instances for v1 clients.

Both the instance converter and the version translator can be generated using the Entity Services API.

To re-extract instances from original source, generate, customize, and install an instance converter based on the new model, as described in “Creating an Instance Converter Module” on page 110. Send your raw source data through the converter, just as you did with the previous model version.

To use a version translator to generate new version instances from old ones, generate, customize, and install a version translator module from the old and new models as described in “Creating a Model Version Translator Module” on page 116. Then, use the translator to convert instance data from the old model to the new one.

The following diagram illustrates using a version translator to generate an envelope document containing an instance based on a new model version. You can also pass just an instance (rather than an envelope document) to the translator.



3.8.2 Replacing the Old Model with a New Version

If all consumers will immediately move to the new model then you can do the following to update your model-based artifacts:

- TDE template, query options, schema artifacts:
  - Generate a version based on the new model.
  - Apply your customizations, including merging in appropriate customizations from the old model.
  - Redeploy the artifacts.
- Database configuration: If the new model adds or removes range indexes and word lexicons, you will need to generate a new configuration artifact, apply your customizations, and update your database configuration.
- Instance converter:
  - Generate a converter based on the new model.
  - Apply your customizations, including merging in appropriate customizations from the old model.
  - Redeploy the module.
- Instance data:
  - Generate instance data based on the new model, as described in “Generating Instances From the New Model” on page 88.
  - Replace the envelope documents based on the old model with the new envelope documents.

Note that you might still be able to serve old version instances to clients by using a down-converting version translator to convert new instances to old ones during extraction. You can generate such a translator using Entity Services; for details, see “Creating a Model Version Translator Module” on page 116.

3.8.3 Making Multiple Model Versions Available

When maintaining multiple model versions, the procedures are similar to those described in “Replacing the Old Model with a New Version” on page 90, but you must consider how to manage multiple versions of your code and configuration artifacts, such as the following:

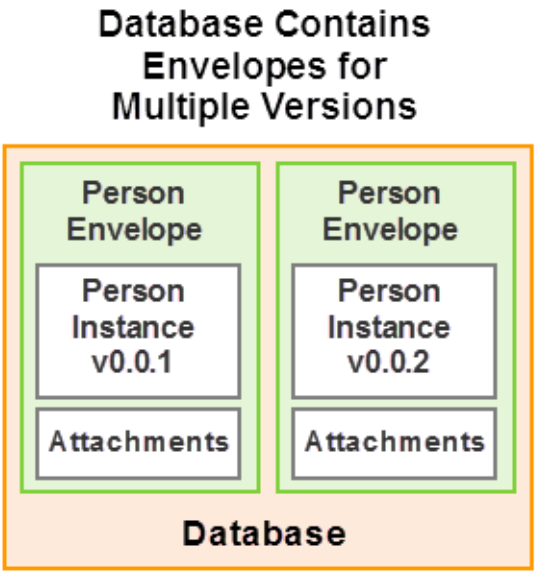
- Instance Data
- Entity Type Schema
- TDE Template
- Query Options
- Database Configuration

3.8.3.1 Instance Data

You must choose an approach to storing your updated instance data in the database. You might use one of the following approaches to managing versions:

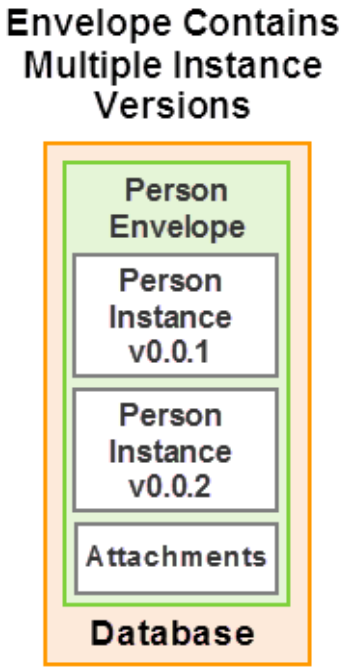
- Each envelope document contains either an old OR a new version of an instance.
- Each envelope document contains both an old AND a new version of an instance.

In the first approach, the database contains envelope documents for instances based on both model versions, as shown in the following diagram:



In this case, putting the envelope documents in different collections based on version will make them easier to manage and search. You can also use the value of `es:instance/es:info/es:version` to distinguish between versions.

In the second approach, the database still contains only one set of envelope documents, but each envelope contains multiple instances, as shown in the following diagram:



You can use the value of `es:instance/es:info/es:version` to distinguish between versions during search and entity extraction. Your instance converter must be customized to store multiple instances in a single envelope.

3.8.3.2 Entity Type Schema

This topic refers to maintaining more than one version of the schemas generated by the `es:schema-generate` XQuery function or the `es.schemaGenerate` Server-Side JavaScript function.

It is usually best to avoid multiple schemas for the same type name. Schema validation is based on type name, so if you do not explicitly specify which schema to use for validation you won't know which schema is applied.

During explicit validation in XQuery, you can import a schema into your evaluation context. For example, if you have v1.0.0 and v2.0.0 schemas installed for a model that defines a `Person` entity type, then you could force validation against the v2.0.0 model by doing the following:

```
xquery version "1.0-ml";
import module namespace es = "http://marklogic.com/entity-services"
  at "/MarkLogic/entity-services/entity-services.xqy";
import schema default element namespace ""
  at "/es-gs/person-2.0.0.xsd";

xdmp:validate(
  es:instance-xml-from-document(
```

```
    fn:doc('/es-gs/envelopes/1234.xml')),
'type', xs:QName('PersonType'))
```

For XML instance representations, you can add `@schemaLocation` to control which schema is applied. For more details, see Referencing Your Schema in the *Application Developer's Guide*.

3.8.3.3 TDE Template

The triples generated from a TDE template generated by Entity Services use a subject IRI that includes the model version. Therefore, there is no collision between the facts generated from each template version.

However, both templates will use the same row `schema-name` for the same entity types, which will cause row searches to return the union of matched by both templates. To avoid this, you should give each entity type row schema a unique name.

3.8.3.4 Query Options

You can merge old and new version query options together, or keep them separate and use the version appropriate for entity instance versions you're searching.

If you choose to keep multiple versions of canonical instances in a single envelope document, you should probably modify your query options to include version related constraints or additional queries.

For example, you might want to add a version constraint based on `es:envelope/es:instance/es:info/es:version`.

3.8.3.5 Database Configuration

The database configuration is single-state. You can configure the union of range indexes and word lexicons defined by the two models.

You should usually not remove a range index or word lexicon required by the older model if you wish to continue supporting searches on that version. Also, if you define a range index or word lexicon for a property that exists in both model versions, you might see different search results against the old version entities because queries against the shared property can now be resolved out of the index.

3.9 Model Descriptor Syntax Reference

This section provides a detailed description of the layout of a model descriptor, including syntax, component descriptions, and examples. A model descriptor has the following top level structure, where the `info` section contains model metadata, and the `definitions` section contains your entity type definitions. A model descriptor must define at least one entity type.

JSON	XML
{ "info": model_info, "definitions": { entity_type_definition, ... } }	<es:model xmlns:es="http://marklogic.com/entity-services"> <es:info>model_info</es:info> <es:definitions> entity_type_definition ... <es:definitions> </es:model>

To explore the component parts of a model descriptor in more detail, see the following topics:

- model_info
- entity_type_definition
- property_definition

3.9.1 model_info

The “info” section of a model descriptor contains model metadata, such as a description or version.

- Syntax Summary
- Component Description
- Examples

3.9.1.1 Syntax Summary

The “info” section of a model descriptor has the following structure:

JSON	XML
{ "title": string, "version": string, "baseUri": string, "description": string }	<es:info xmlns:es="http://marklogic.com/entity-services"> <es:title>*model title*</es:title> <es:version>*model version*</es:version> <es:base-uri>*absolute uri*</es:base-uri> <es:description>*model desc*</es:description> </es:info>

3.9.1.2 Component Description

The “info” section of a model descriptor can contain the following XML elements or JSON properties. Title and version are the only required items.

Property Name	Description
`title`	Required. The title of this model descriptor. The title string must be a valid XQuery namespace prefix. If you plan to generate a TDE template from the model, you should avoid using hyphens (“-”) in the title. Hyphens will be converted to underscores (“_”) in the TDE schema, view, and column names, in order to avoid invalid SQL names. The title is used as the module namespace prefix when generating code from the model. If the first character of the title is upper case, it will be converted to lower space when used as namespace prefix.
`version`	Required. The version of this model descriptor. Best practice is to use the “semver” format, such as “1.0.0”; for details, see http://semver.org/. The version number of the model is considered the version number of all the entity types defined within the model.
`baseUri (JSON)` `base-uri (XML)`	Optional. A valid absolute URI, usable to resolve RDF values in the descriptor. If this entity property is not present, `http://example.org/` is used as the default URI. For details, see “Controlling the Model IRI and Module Namespaces” on page 79.
`description`	Optional. A description of this set of entity type definitions. This is purely information metadata.

3.9.1.3 Examples

The following example contains an `info` section that uses all available properties. Only the `title` and `version` properties are required.

Format	Example Model Descriptor

JSON	`{ "info": {`
	`  "title": "Example",`
	`  "description": "ES Examples",`
	`  "version": "1.0.0",`
	`  "baseUri": "http://es-ex/examples",`
	`  },`
	`  "definitions": {`
	`  "Person": {`
	`  "properties": {`
	`  "id": { "datatype": "int" },`
	`  "name": { "datatype": "string" }`
	`} } } }`

XML	`<es:model xmlns:es="http://marklogic.com/entity-services">`
	`  <es:info>`
	`  <es:title>Example</es:title>`
	`  <es:description>ES Examples</es:description>`
	`  <es:version>1.0.0</es:version>`
	`  <es:base-uri>http://es-ex/examples</es:base-uri>`
	`  </es:info>`
	`  <es:definitions>`
	`  <Person>`
	`  <es:properties>`
	`  <id><es:datatype>int</es:datatype></id>`
	`  <name><es:datatype>string</es:datatype></name>`
	`  </es:properties>`
	`  </Person>`
	`  </es:definitions>`
	`</es:model>`

3.9.2 entity_type_definition

An entity type definition is a child of the “definitions” section of a model descriptor. A model descriptor must include at least one entity type definition, and may contain multiple entity type definitions.

3.9.2.1 Syntax Summary

An entity type definition has the following structure, where *entityTypeName* (in JSON) and *entity-type-name* (in XML) represent the user-defined entity type name, such as Person or Order. By convention, entity type names begin with a capital letter (“Person”, not “person”).

If you plan to generate a TDE template from the model, you should avoid using hyphens (“-”) in the entity type and entity property names. Hyphens will be converted to underscores (“_”) in the TDE schema, view, and column names, in order to avoid invalid SQL names.

JSON	XML

entityTypeName : {
  "properties": {
  propertyName: property_definition,
  ...
  },
  "required": [*string*],
  "primaryKey": *string*,
  "namespace": *string*,
  "namespacePrefix": *string*,
  "pii": [*string*],
  "pathRangeIndex": [*string*],
  "elementRangeIndex": [*string*],
  "rangeIndex": [*string*],
  "wordLexicon": [*string*]
  "description": *string*
 	
}
	<*entity-type-name*
	  xmlns:es="http://marklogic.com/entity-services">
	  <es:properties>
	  <*property-name*>
	  property_definition
	  </*property-name*>
	  ...
	  </es:properties>
	  <es:required>*property name*</es:required>
	  <es:primary-key>
	  property name
	  </es:primary-key>
	  <es:namespace>*namespace URI*</es:namespace>
	  <es:namespace-prefix>
	  namespace prefix
	  </es:namespace-prefix>
	  <es:pii>*property name*</es:pii>
	  <es:path-range-index>
	  property name
	  </es:path-range-index>
	  <es:element-range-index>
	  property name
	  </es:element-range-index>
	  <es:range-index>
	  property name
	  </es:range-index>
	  <es:word-lexicon>
	  property name
	  </es:word-lexicon>
	  <es:description>*type desc*</es:description>
	</*entity-type-name*>

3.9.2.2 Component Description

An entity type definition can contain the following XML elements or JSON properties.

Property Name	Description
`properties`	Optional. Zero or more entity property definitions. Each child JSON property or XML element name is the name of a property of the entity type. In XML, the element name must not be namespace qualified. For more details, see “Writing a Model Descriptor” on page 59.
`description`	Optional. A description of this entity type.
`required`	Optional. Specify the names of entity properties that must be in every instance of this entity type. In XML, include multiple `required` elements to specify multiple required property names. Each named entity property must match the name of an entity property defined in the `properties` section of this entity type definition. Any entity properties not tagged as required are treated as optional. For more details, see “Distinguishing Required and Optional Entity Properties” on page 70.
`primaryKey (JSON)` `primary-key (XML)`	Optional. The name of an entity property to use as a primary key when generating artifacts such as an extraction template. The value must match the name of an entity property defined in the `properties` section of this entity type definition. There can be at most one primary key in an entity type definition. The primary key property is implicitly also a required property. For more details, see “Identifying the Primary Key Entity Property” on page 67.
`namespace`	Optional. A namespace URI with which to qualify canonical XML entity instances of this type. If you include a namespace URI, you must also define a namespace prefix using the `namespace-prefix` XML element or `namespacePrefix` JSON property. The namespace is also used in generated database configuration and query options artifacts. For details and restrictions, see “Defining a Namespace URI for an Entity Type” on page 71.

Property Name	Description
`namespacePrefix (JSON)` `namespace-prefix (XML)`	Optional. A namespace prefix to bind to the XML namespace defined by the `namespace` XML element or JSON property. You must define a prefix if you define a namespace. For details and restrictions, see "Defining a Namespace URI for an Entity Type" on page 71.
`pii`	Optional. The name(s) of entity properties that can contain Personally Identifiable Information (PII). You can generate an Element Level Security (ELS) configuration to more tightly restrict acess to PII properties than access to other instance properties. For details, see "Identifying Personally Identifiable Information (PII)" on page 69. In XML, include multiple `pii` elements to specify multiple properties.
`pathRangeIndex (JSON)` `path-range-index (XML)`	Optional. The name(s) of entity properties that should be backed by a path range index. This affects the database configuration and query options you can generate from a model. Each named property must match the name of an entity property defined in the `properties` section of this entity type definition. In XML, include multiple `path-range-index` elements to specify multiple properties. For more details, see "Identifying Entity Properties for Indexing" on page 75.
`elementRangeIndex (JSON)` `element-range-index (XML)`	Optional. The name(s) of entity properties that should be backed by an element range index. This affects the database configuration and query options you can generate from a model. Each named property must match the name of an entity property defined in the `properties` section of this entity type definition. In XML, include multiple `element-range-index` elements to specify multiple properties. For more details, see "Identifying Entity Properties for Indexing" on page 75.
`rangeIndex (JSON)` `range-index (XML)`	Optional. Deprecated. Equivalent to the `pathRangeIndex` property in a JSON descriptor, or the `path-range-index` element in an XML descriptor.
`wordLexicon (JSON)` `word-lexicon (XML)`	Optional. The name(s) of entity properties that should be backed by a word lexicon. This affects the database configuration and query options you can generate from a model. Each named property must match the name of an entity property defined in the `properties` section of this entity type definition. In XML, include multiple `word-lexicon` elements to specify multiple properties. For details, see "Identifying Entity Properties for Indexing" on page 75.

3.9.2.3 Examples

The following example defines a Person entity type that contains entity properties named “id”, “name”, “bio”, and “rating”. The “id” and “name” properties are required. The “id” entity property is a primary key. A path range index is required for “id” and “rating”, and a word lexicon is required for “bio”. The “name” property is tagged as PII.

Format	Example Model Descriptor

JSON	`{ "info": {`
	`  "title": "Example",`
	`  "description": "ES Examples",`
	`  "version": "1.0.0"`
	`  },`
	`  "definitions": {`
	`  "Person": {`
	`  "properties": {`
	`  "id": { "datatype": "int" },`
	`  "name": { "datatype": "string" },`
	`  "bio": { "datatype": "string" },`
	`  "rating": { "datatype": "float" }`
	`  },`
	`  "required": ["id", "name"],`
	`  "primaryKey": "id",`
	`  "pii": ["name"],`
	`  "pathRangeIndex": ["id", "rating"],`
	`  "wordLexicon": ["bio"],`
	`  "namespace": "http://example.org/es/gs",`
	`  "namespacePrefix": "es"`
	`}}}`

Format	Example Model Descriptor
XML	<es:model xmlns:es="http://marklogic.com/entity-services"> <es:info> <es:title>Example</es:title> <es:description>ES Examples</es:description> <es:version>1.0.0</es:version> </es:info> <es:definitions> <Person> <es:properties> <id><es:datatype>int</es:datatype></id> <name><es:datatype>string</es:datatype></name> <bio><es:datatype>string</es:datatype></bio> <rating><es:datatype>float</es:datatype></rating> </es:properties> <es:required>id</es:required> <es:required>name</es:required> <es:primary-key>id</es:primary-key> <es:pii>name</es:pii> <es:path-range-index>id</es:path-range-index> <es:path-range-index>rating</es:path-range-index> <es:word-lexicon>bio</es:word-lexicon> <es:namespace>http://example.org/es/gs</es:namespace> <es:namespace-prefix>esgs</es:namespace-prefix> </Person> </es:definitions> </es:model>

3.9.2.4 See Also

For more details about using this component, see the following topics:

- “Writing a Model Descriptor” on page 59

3.9.3 property_definition

An entity property definition is a child of the entity_type_definition section of a model descriptor. Each entity type must include at least one entity property definition.

- Syntax Summary
- Component Description
- Examples
- See Also

3.9.3.1 Syntax Summary

An entity property definition can have one of the following forms. Entity property definition are used in the `properties` child of an entity_type_definition.

JSON	XML
{ "datatype" : "string", "collation": *string*, "description": *string* }	<!-- string-valued entity property --> <es:datatype>string</es:datatype> <es:collation> *collationUri* </es:collation> <es:description>*desc*</es:description>
{ "datatype" : "array", "items": property_definition , "description": *string* }	<!-- array/list-valued property --> <es:datatype>array</es:datatype> <es:items>property_definition</es:items>
{ "datatype" : property_type, "description": *string* }	<!-- prop of any other type --> <es:datatype>property_type</es:datatype> <es:description>*desc*</es:description>
{ "$ref": *string*, "description": *string* }	<!-- ref to another entity type --> <es:ref>*type path ref*</es:ref> <es:description>*desc*</es:description>

3.9.3.2 Component Description

This portion of a model descriptor can contain the following XML elements or JSON properties. An entity property definition must include either a `datatype` or `ref` JSON property or XML element, but not both.

Property Name	Description
`datatype`	Required if `$ref` (JSON) or `es:ref` (XML) is not present. The data type of values in this entity property. The value must be one of the types listed in "property_type" on page 105. The datatype can affect what other JSON properties or XML elements can be included in this definition, such as a datatype of `string` enabling the inclusion of a collation URI in the property definition.
`$ref` (JSON) `ref` (XML)	Required if `datatype` is not present. A reference to another entity type, in the form of either a relative path to an entity type defined in this model descriptor or an absolute IRI. The value must end in a simple type name so that it can be treated as a type name during code generation. For details, see "Defining Entity Relationships" on page 80 and "Defining an Entity Property with a Complex Type" on page 65.
`collation`	Optional. Only usable when the value of `datatype` is `string`. The collation to use when generating index/lexicon configuration and query options. If you do not specify a collation, then it defaults to `http://marklogic.com/collation/en`.
`items`	Required when the value of `datatype` is `array`. The type definition for the array items. The value is itself an entity_type_definition. For details, see "Defining an Entity Property with Array Type" on page 66.
`description`	Optional. A description of this entity type.

3.9.3.3 Examples

The following example defines a Person entity type with 3 entity properties: An “id” of type “int”, a “name” with type string whose definition includes a collation, and a “friend” entity property with array type. Each item value in the “friend” array is a reference to a `Person` entity.

Format	Example Model Descriptor

JSON	`{ "info": {`
	`  "title": "Example",`
	`  "description": "ES Examples",`
	`  "version": "1.0.0"`
	`},`
	`"definitions": {`
	`  "Person": {`
	`  "properties": {`
	`  "id": { "datatype": "int" },`
	`  "name": {`
	`  "datatype": "string",`
	`  "collation": "http://marklogic.com/collation/"`
	`  },`
	`  "friend": {`
	`  "datatype" : "array",`
	`  "items": { "$ref" : "#/definitions/Person" }`
	`  }`
	`}}}}`

XML	`<es:model xmlns:es="http://marklogic.com/entity-services">`
	`<es:info>`
	`<es:title>Example</es:title>`
	`<es:description>ES Examples</es:description>`
	`<es:version>1.0.0</es:version>`
	`</es:info>`
	`<es:definitions>`
	`<Person>`
	`<es:properties>`
	`<id><es:datatype>int</es:datatype></id>`
	`<name>`
	`<es:datatype>string</es:datatype>`
	`<es:collation>http://marklogic.com/collation/</es:collation>`
	`</name>`
	`<friend>`
	`<es:datatype>array</es:datatype>`
	`<es:items>`
	`<es:ref>#/definitions/Person</es:ref>`
	`</es:items>`
	`</friend>`
	`</es:properties>`
	`</Person>`
	`</es:definitions>`
	`</es:model>`

3.9.3.4 See Also

For more details, see the following topics:

- “Writing a Model Descriptor” on page 59

3.9.4 property_type

This section defines the type names that can be specified in the `datatype` JSON property or XML element of an entity property definition. With the exception of “iri” and “array”, these types correspond to XML Schema Definition Language (XSD) of the same name; for details, see https://www.w3.org/TR/xmlschema11-2/#built-in-datatypes.

iri	duration	negativeInteger
array	float	nonNegativeInteger
anyURI	gDay	nonPositiveInteger
base64Binary	gMonth	short
boolean	gMonthDay	string
byte	gYear	time
date	gYearMonth	unsignedByte
dateTime	hexBinary	unsignedInt
dayTimeDuration	int	unsignedLong
decimal	integer	unsignedShort
double	long	yearMonthDuration

Note: Not all these datatypes are usable as range index or word lexicon types. If you specify an entity property with an incompatible type in the range index or word lexicon specification of an entity type definition, then the resulting model will not validate.

An array-typed entity property contains an item type definition that also uses this type list. For details, see “property_definition” on page 101 and “Defining an Entity Property with Array Type” on page 66.

Some types are folded into a compatible super-type when defining range indexes. For example, entity properties of type “iri” are indexed as “string”, and entity properties of type “byte” or “short” are indexed as “int”. Some data type cannot be used for index or word lexicon configuration.

For more details, see the following topics:

- “property_definition” on page 101
- “Writing a Model Descriptor” on page 59
- “Identifying Entity Properties for Indexing” on page 75
- “Supported Datatypes” on page 78 (about type restrictions on indexing)

4.0 Generating Code and Other Artifacts

The Entity Services API includes tools for generating code templates and configuration artifacts that enable you to quickly bring up a model-based application.

For example, you can generate code for creating instances and instance envelope documents from raw source and converting instances between different versions of your model. You can also generate an instance schema, TDE template, query options, and database configuration based on a model.

This chapter covers the following topics:

4.1 Code and Artifact Generation Overview

The following steps outline the basic procedure for generating code and configuration artifacts using the Entity Services API. The specifics are described in detail in the rest of this chapter.

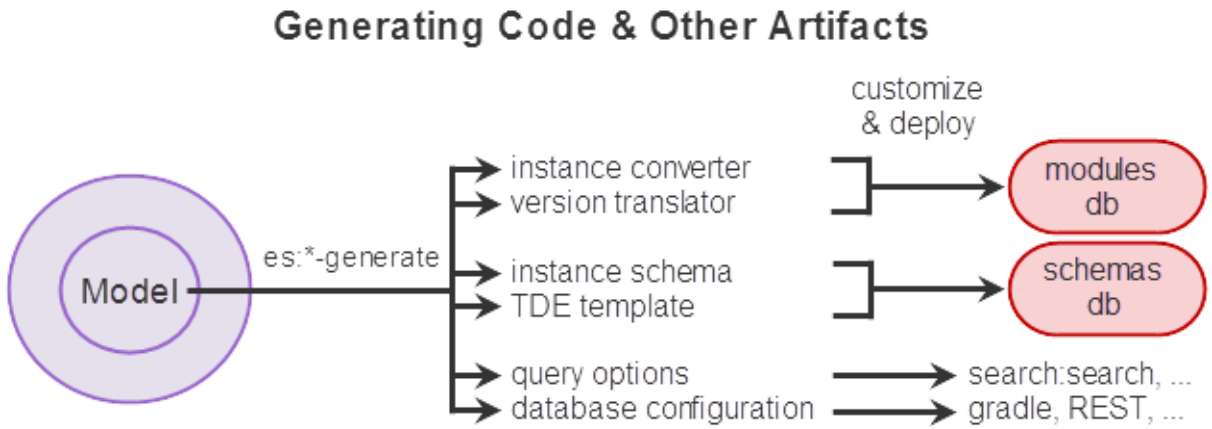
1. Author a model descriptor and create a model, as described in “Creating and Managing Models” on page 57.

2. Pass the model (in the form of a `json:object` or JSON `object-node`) to one of the `es:*-generate` XQuery functions or `es.*Generate` JavaScript functions to generate a code module or configuration artifact.

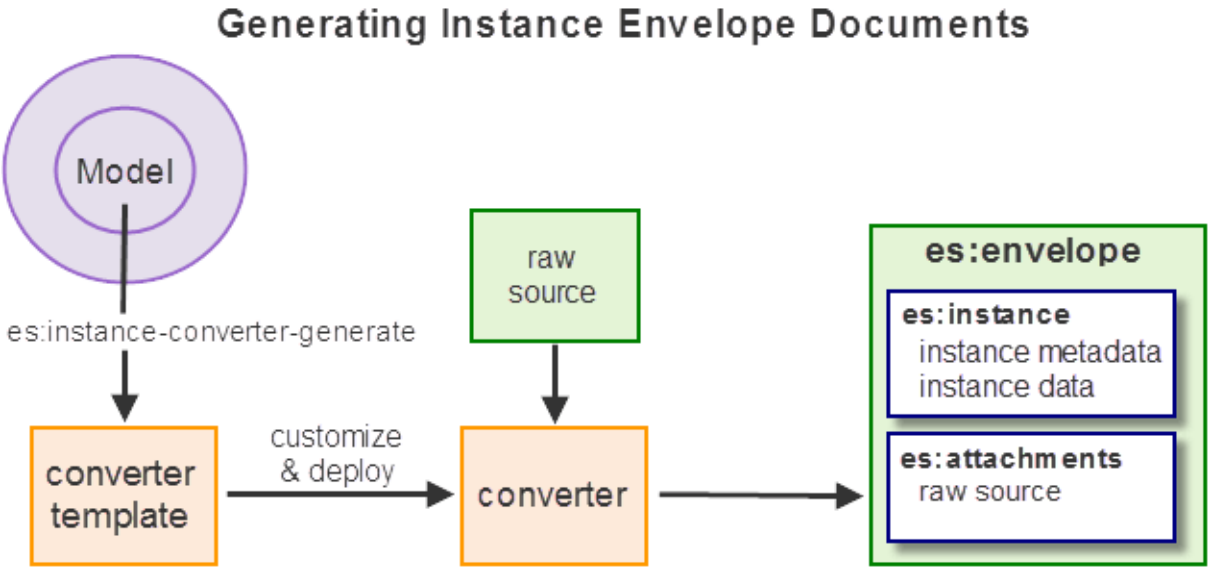
3. Customize the generated code or artifact to meet the needs of your application. All generated code and artifacs are usable as-is, but you will want to customize some of them.

4. Deploy the (customized) code or artifact, as needed. Code modules must be deployed to the modules database. Artifacts such as the TDE template must be deployed to the schemas database. Artifacts such as query options do not require deployment.

The following diagram illustrates this process. The relevant part of the model is the portion represented by the model descriptor.



The following diagram illustrates the high level flow for creating, deploying and using an instance converter module. The instance converter module is discussed in more detail in “Creating an Instance Converter Module” on page 110.



4.2 Summary of Available Generators

The following table summarizes the code and artifact generation functions provided by Entity Services. Both the XQuery (`es:*`) and Server-Side JavaScript (`es.*`) name of each function is included. For more details, see the *MarkLogic XQuery and XSLT Function Reference* or *MarkLogic Server-Side JavaScript Function Reference*.

Function	Description

`es:instance-converter-generate`
 	
`es.instanceConverterGenerate`
	Generate an XQuery library module containing functions useful for data conversion, such as converting raw source data into entity instances or an instance into its canonical representation. You can use this module from either XQuery or Server-Side JavaScript. For more details, see "Creating an Instance Converter Module" on page 110.

`es:version-translator-generate`
 	
`es.versionTranslatorGenerate`
	Generate an XQuery library module containing functions useful for converting entity instances from one version to another. You can use this module from either XQuery or Server-Side JavaScript. For more details, see "Creating a Model Version Translator Module" on page 116.

`es:schema-generate`
 	
`es.schemaGenerate`
	Generate an XSD schema for a model. The resulting schema is suitable for validating canonical entity instances. For details, see "Generating an Entity Instance Schema" on page 129.

`es:extraction-template-generate`
 	
`es.extractionTemplateGenerate`
	Generate a TDE template that facilitates searching entity instances as row or semantic data. For more deails, see "Generating a TDE Template" on page 122.

`es:database-properties-generate`
 	
`es.databasePropertiesGenerate`
	Generate a JSON database properties configuration object, suitable for use with the REST Management API or `ml-gradle`. This artifact includes range index and word lexicon configuration based on the model descriptor. For details, see "Generating a Database Configuration Artifact" on page 137.

Function	Description

`es:search-options-generate`
 	
`es.searchOptionsGenerate`
	Generates a set of query options helpful for searching entity instances with the XQuery Search API or the REST, Java, or Node.js Client APIs. For more details, see "Generating Query Options for Searching Instances" on page 141.

`es:pii-generate`
 	
`es.piiGenerate`
	Generate an Element Level Security configuration artifact that enables stricter control of entity properties that contain Personally Identifiable Information (PII). For more details, see "Generating a PII Security Configuration Artifact" on page 134.

4.3 Creating an Instance Converter Module

An instance converter helps you create entity instance documents from raw source data. Generate a default instance converter using Entity Services, then customize the code for your application.

4.3.1 Purpose of a Converter Module

An instance converter is a key part of a model-driven application. The instance converter provides functions that facilitate the following tasks:

- Creating an entity instance from raw source data.
- Creating an entity envelope document that encapsulates an instance, metadata, and raw source data.
- Extracting a canonical instance or its attachments (such as the raw source) from an envelope document.

For more details on envelope documents, see "What is an Envelope Document?" on page 150.

You usually use the instance converter to create entity instance envelope documents and to extract canonical instances for use by downstream entity consumers.

You are expected to customize the generated converter module to meet the needs of your application.

4.3.2 Generating a Converter Module Template

Generate an instance converter from the JSON `object-node` or `json:object` representation of a model descriptor by calling the XQuery function `es:instance-converter-generate` or the JavaScript function `es.instanceConverterGenerate`. The result is an XQuery library module containing both model-specific and entity type specific code.

The input to the generator is a JSON descriptor. If you have an XML descriptor, you must first convert it to the expected format; for details, see “Working With an XML Model Descriptor” on page 84. The output of the generator function is an XQuery library module.

You can use the generated code as-is, but most applications will require customization of the converter implementation. For details, see “Customizing a Converter Module” on page 114.

The following example code generates a converter module from a previously persisted descriptor, and then saves the generated code as a file on the filesystem.

Language	Example

XQuery	`xquery version "1.0-ml";`
	`import module namespace es =`
	`  "http://marklogic.com/entity-services"`
	`  at "/MarkLogic/entity-services/entity-services.xqy";`
	
	`let $desc := fn:doc('/es-gs/models/person-1.0.0.json')`
	`return xdmp:save(`
	`  '/space/es/gs/person-1.0.0-conv.xqy',`
	`  es:instance-converter-generate($desc)`
	`)`

JavaScript	`'use strict';`
	`const es = require('/MarkLogic/entity-services/entity-services.xqy');`
	
	`const desc = cts.doc('/es-gs/models/person-1.0.0.json');`
	`xdmp.save(`
	`  '/space/es/gs/person-1.0.0-conv.xqy',`
	`  es.instanceConverterGenerate(desc)`
	`);`

You could also insert the converter directly into the modules database, but the converter is an important project artifact that should be placed under source control. You will want to track changes to it as your application evolves.

4.3.3 Understanding the Default Converter Implementation

This section explores the default code generated for an instance converter module. The following topics are covered:

- Module Namespace Declaration

- Generated Functions

4.3.3.1 Module Namespace Declaration

The generated module begins with a module namespace declaration of the following form, derived from the `info` section of the model.

```
module namespace normalizedTitle = "baseUri/title-version";
```

For example, if your descriptor contains the following metadata:

```
"info": {
  "title": "Example",
  "version": "1.0.0",
  "baseUri": "http://marklogic.com/examples/"
}
```

Then the converter module will contain the following module namespace declaration. Notice that the leading upper case letter in the `title` value (“Example”) is converted to lower case when used as a namespace prefix.

```
module namespace example =
  "http://marklogic.com/examples/Example-1.0.0";
```

If the model `info` section does not include a `baseUri` setting, then the namespace declaration uses the base URI “http://example.org/”.

If the `baseUri` does not end in a forward slash (“/”), then the module namespace URI is relative. For example, if `baseUri` in the previous example is set to “http://marklogic.com/example”, then the module namespace declaration is as follows:

```
module namespace example =
  "http://marklogic.com/examples#Example-1.0.0";
```

To learn more about the base URI, see “Controlling the Model IRI and Module Namespaces” on page 79.

4.3.3.2 Generated Functions

The converter module implements the following public functions, plus some private utility functions for internal use by these functions.

Function	Description
ns:extract-instance-*T*	Transform raw source data into an in-memory entity instance. One such function is generated for each entity type *T* defined by the model descriptor. This function produces a *T* instance as a `json:object` (a special type of `map:map`).
ns:instance-to-envelope	Create an entity envelope document from an entity instance. You will not usually need to customize this function. The input to this function is an entity instance of the form produced by *ns*:extract-instance-*T*.
ns:instance-to-canonical	Create the canonical XML or JSON representation of an entity instance from the `json:object` representation. You will not usually call this function directly or customize it. Rather, the *ns*:instance-to-envelope function uses it internally.

Each `extract-instance-T` function is a starting place for synthesizing an entity instance from raw source data. These functions are where you will apply most of your customizations to the generated code.

The input to an `extract-instance-T` function is a node containing the source data. The output is an entity instance represented as a `json:object`. By default, the instance encapsulates a canonicalized entity with the original source document. This is default envelope document repesentation.

In pseudo code, the generated implementation is as follows:

```
declare function ns:extract-instance-T(
  $source-node as node()
) as map:map
{
  normalize the input source reference
  initialize variables for the values of each entity property
  initialize an empty instance of type T
  attach the source data to the instance
  assign values to the instance properties
};
```

The portion of the function that sets up the entity property values is where you will apply most or all of your customizations. The default implementation assumes a one-to-one mapping between source and entity instance property values.

For example, suppose the model contains a “Person” entity type, with entity properties “firstName”, “lastName”, and “fullName”. Then the default `extract-instance-Person` implementation contains code similar to the following. The section following the “begin customizations here” comment is where you make most or all of your customizations.

```
declare function example:extract-instance-Name(
    $source-node as node()
) as map:map
{
    let $source-node := es:init-source($source, 'Person')
    (: begin customizations here :)
    let $id  := $source-node/id ! xs:string(.)
    let $firstName  := $source-node/firstName ! xs:string(.)
    let $lastName  := $source-node/lastName ! xs:string(.)
    let $fullName  := $source-node/fullName ! xs:string(.)
    (: end customizations :)

    let $instance := es:init-instance($source-node, 'Person')
    (: Comment or remove the following line to suppress attachments :)
        =>es:add-attachments($source)

    return
    if (empty($source-node/*))
    then $instance
    else $instance
        =>   map:with('id', $id)
        =>   map:with('firstName', $firstName)
        =>   map:with('lastName', $lastName)
        =>   map:with('fullName', $fullName)
};
```

If the source XML elements or JSON objects have different names or require a more complex transformation than a simple type cast, customize the implementation. For more details, see “Customizing a Converter Module” on page 114.

Comments in the generated code describe the default implementation in more detail and provide suggestions for common customizations.

4.3.4 Customizing a Converter Module

Most customization involves changing the portion of each *ns*`:extract-instance-`*T* function that sets the values of the instance properties.

The default implementation of this portion of an extract function assumes that some property *P* in the entity instance gets its value from a node of the same name in the source data, and that a simple typecast is sufficient to convert the source value to the instance property type defined by the model.

For example, if an entity type named `Person` defines a string-valued property named `firstName`, then the generated code in `firstName` in `example:extract-instance-Person` related to intializing this property looks like the following:

```
let $firstName  := $source-node/firstName ! xs:string(.)
...
let $instance := es:init-instance($source-node, 'Person')
....
if (empty($source-node/*))
then $instance
else $instance
        ...
        =>   map:with('firstName', $firstName)
        ...
```

You might need to modify the code to perform a more complex transformation of the value, or extract the value from a different location in the source node. For example, if your source data uses the property name “given” to hold this information, then you would modify the generated code as follows:

```
let $firstName  := $source-node/given ! xs:string(.)
```

The following list describes other common customization use cases:

- Synthesize a property value from other data. For example, aggregate an instance property from other values in your source data, or extract a value from other sources, based on information in the source node.
- Normalize data formats. For example, data such as dates, telephone numbers, and social security numbers often occur in multiple formats in raw data. You can normalize such data to a single format in your instances for easy search and comparison.
- Assign a default value for missing data. If you know that a required property in your entity instance is not always present in your source data, you can modify the code to ensure the entity instance contains a reasonable default value.

Once you finish customizing the code, you must deploy the module to your App Server before you can use the code. For details, see “Deploying Generated Code and Artifacts” on page 147.

For a more complete example, see “Getting Started With Entity Services” on page 19 or the Entity Services examples on GitHub. For details on locating the GitHub examples, see “Exploring the Entity Services Open-Source Examples” on page 15.

4.4 Creating a Model Version Translator Module

You can use the Entity Services API to generate a template for transitioning entity instance data from one version of your model to another. This section covers the following topics:

- Purpose of a Version Translator
- Generating a Version Translator Module Template
- Understanding the Default Version Translator Implementation

For an end-to-end example of handling model version changes, see the Entity Services examples on GitHub. For more details, see “Exploring the Entity Services Open-Source Examples” on page 15.

4.4.1 Purpose of a Version Translator

A version translator is an XQuery library module that helps you convert instance data conforming to one model version into another.

The version translator only addresses instance conversion. Model changes can also require changes to other artifacts, such as the TDE template, schema, query options, instance converter, and database configuration. For more details, see “Managing Model Changes” on page 88.

Though you can run the generated translator code as-is, it is meant to serve as a starting point for your customizations. Depending on the ways in which your source and target models differ, you might be required to modify the code.

4.4.2 Generating a Version Translator Module Template

Generate a version translator using the XQuery function `es:version-translator-generate` or the JavaScript function `es.versionTranslatorGenerate`. The output is an XQuery library module that you can customize and install in your modules database.

The inputs to the generator are source and target model descriptors, as JSON. If you have an XML descriptor, you must first convert it to the expected format; for details, see “Working With an XML Model Descriptor” on page 84.

You can use the generated code as-is, but most applications will require customization of the converter implementation. For details, see “Customizing a Version Translator Module” on page 119.

You must install the translator module in your modules database before you can use it. For details, see “Deploying Generated Code and Artifacts” on page 147.

The following example code generates a translator module from previously persisted descriptors, and then saves the generated code as a file on the filesystem. The resulting module is designed to convert instances of version 1.0.0 to instances of version 2.0.0.

Language	Example

XQuery	`xquery version "1.0-ml";`
	`import module namespace es =`
	`  "http://marklogic.com/entity-services"`
	`  at "/MarkLogic/entity-services/entity-services.xqy";`
	
	`let $v1 := fn:doc('/es-gs/models/person-1.0.0.json')`
	`let $v2 := fn:doc('/es-gs/models/person-2.0.0.json')`
	`return xdmp:save(`
	`  '/space/es/gs/models/person-1.0.0-to-2.0.0.xqy',`
	`  es:version-translator-generate($v1,$v2)`
	`)`

JavaScript	`'use strict';`
	`const es = require('/MarkLogic/entity-services/entity-services.xqy');`
	
	`const v1 = cts.doc('/es-gs/models/person-1.0.0.json');`
	`const v2 = cts.doc('/es-gs/models/person-2.0.0.json');`
	`xdmp.save(`
	`  '/space/es/gs/models/person-1.0.0-to-2.0.0.xqy',`
	`  es.versionTranslatorGenerate($v1,$v2)`
	`);`

You could also insert the translator directly into the modules database, but the translator is an important project artifact that should be placed under source control. You will want to track changes to it as your application evolves.

4.4.3 Understanding the Default Version Translator Implementation

This section explores the default code generated for a version translator module. This information can help guide your customizations. This section covers the following topics:

- Module Namespace Declaration
- Generated Functions
- Customizing a Version Translator Module

4.4.3.1 Module Namespace Declaration

The generated module begins with a module namespace declaration of the following form, derived from the `info` section of the two models.

```
module namespace title2-from-title1 =
  "baseUri2/title2-version2-from-title1-version1";
```

Where *title1* and *version1* come from the `info` section of the source model, *title2* and *version2* come from the `info` section of the target model, and *baseUri2* comes from the `info` section of the target model. (The base URI from the source model is unused.) The titles are normalized to all lower case.

For example, suppose the source and target models contain the following info sections, reflecting a change from version 1.0.0 to version 2.0.0 of a model with the title “Person”. The model title is unchanged between versions.

Model	Info Section

Source	`"info": {`
	`  "title": "Person",`
	`  "version": "1.0.0",`
	`  "baseUri": "http://example.org/example-person/"`
	`}`

Target	`"info": {`
	`  "title": "Person",`
	`  "version": "2.0.0",`
	`  "baseUri": "http://example.org/example-person/"`
	`}`

Then the version translator module will contain the following module namespace declaration.

```
module namespace person-from-person
  = "http://example.org/example-person/Person-2.0.0-from-Person-1.0.0";
```

If the `info` section of the target model does not include a `baseUri` setting, then the namespace declaration uses the base URI “http://example.org/”.

If the target `baseUri` does not end in a forward slash (“/”), then the module namespace URI is relative. For example, if `baseUri` in the previous example has no trailing slash, then the module namespace declaration is as follows:

```
module namespace person-from-person
  = "http://example.org/example-person#Person-2.0.0-from-Person-1.0.0";
```

4.4.3.2 Generated Functions

The version translator module contains a translation function named *ns*`:convert-instance-`*T* for each entity type *T* defined in the target model. The module can contain additional functions, but these for internal use by the translator module. The `convert-instance-`*T* functions are the “public” face of the converter.

For example, if the target model defines a `Name` entity type and a `Person` entity type and the title of the both the source and target model is `Person`, then the generated translation module will contain the following functions:

- `person-from-person:convert-instance-Name`
- `person-from-person:convert-instance-Person`

The input to a `convert-instance-`*`T`* function should be an entity instance or envelope document conforming to the source model version of type *T*. The output is an in-memory instance conforming to the target model version of type T, similar to the output from the `extract-instance-`T function of an instance converter module.

For each entity type property that is unchanged between the two versions, the default `convert-instance-`*`T`* code simply copies the value from source instance to target instance. Actual differences, such as a property that only exists in the target model, require customization of the translator. For details, see "Customizing a Version Translator Module" on page 119.

For an example, see `example-version` in the Entity Services examples on GitHub. To download a copy of the examples, see "Exploring the Entity Services Open-Source Examples" on page 15.

4.4.4 Customizing a Version Translator Module

This section describes some common model changes, how they are handled by the default translation code, and when customizations are likely to be required.

Most of your translator customizations go in the block of variable declarations near the beginning of the conversion function. For example, the block of code shown in bold, below. These declarations set up the values to be assigned to the properties of the new instance, later in the conversion function. The variable names and default initial values are model-dependent.

```
declare function person-from-person:convert-instance-Person(
    $source as node()
) as map:map
{
    let $source-node := es:init-translation-source($source, 'Person')

    let $id := $source-node/id ! xs:string(.)
    let $firstName := $source-node/firstName ! xs:string(.)
    let $lastName := $source-node/lastName ! xs:string(.)
    let $fullName := $source-node/fullName ! xs:string(.)

    return...
```

The table below provides a brief overview of some common entity type definition changes and what customizations they might require. The context for the code snippets is the property value initialization block shown in the previous example. All the code snippets assume a required property; if the property under consideration is optional, then the call to `map:with` would be replaced by a call to `es:optional`.

Use Case	Notes on the Generated Code

Unchanged Property	The default code copies the value of the source instance to the target instance. For array valued properties, the `es:extract-array` utility function performs the copy.
	
	For example, if both source and target contain a property named “thing”, the default translator function includes a line similar to one of the following:
	
	`(: atomic type (string, in this case) :)`
	`let $thing := $source-node/thing ! xs:string(.)`
	
	`(: array type (item type string, in this case) :)`
	`let $thing := ex:extract-array($source-node/thing, xs:string#))`
	
	`(: reference to a locally resolvable "Name" entity of type :)`
	`let $extract-reference-Name := es:init-instance(?, 'Name')`
	`let $thing := $soure-node/thing/* ! $extract-reference-Name(.)`

Property Type Change From One Atomic Type to Another	The default code assumes a simple type cast to the target type is sufficient. Customization is required if the types are not meaninfully convertible this way. For example, if a property named “rating” has string type in the source but float type in the target, then the generated code includes the following: `let $rating := $source-node/rating ! xs:float(.)`

Property Type Change from Atomic to Array Type	The default code constructs an array containing a single item that is the value from the source property. This is done by `es:extract-array`. Customization is required if the source and target value types differ and are not meaninfully convertible by a simple type cast.
	
	For example, if the “rating” property is a simple string value in the source, but an array of float values in the target, then the generated code contains the following:
	
	`let $rating :=`
	`  es:extract-array($source-node/rating, xs:float#1))`

Use Case	Notes on the Generated Code

Property Type Change From Array to Atomic Type	The default code populates the target instance with the value from the first item in the source array. A simple type cast is used to convert the value; customization is required if the source and target value types differ and are not meaninfully convertible this way.
	
	For example, if the "rating" property is an array of float values in the source and a single string value in the target, then you see:
	
	`(: Warning: potential data loss, truncated array.  :)`
	`let $rating := xs:string( fn:head($source-node/rating) )`

Property Type Change From Atomic to Local Reference Type	The default code creates a reference from the source value. Since the source value is not an entity, customization is required to construct a meaningful reference. For example, if a property named "name" is a string in the source but a locally resolvable reference to a `Name` entity type in the target, then following is the default translation code: `let $name := $source-node/name ! es:init-instance(?, 'Name')(.)`

Property in Target Only	The default code copies the value from the source instance to the target instance. However, the source instance probably doesn't contain this property, so customization is usually required. You might modify the code to assign a meaningful default value or extract the new value from the raw source in the attachments of the source envelope.
	
	For example, if only the target contains a float typed property named "rating", then the generated code includes the following:
	
	`(: The following property was missing from the source type.`
	`   The XPath will not up-convert without intervention.  :)`
	`let $rating := $source-node/rating ! xs:float(.)`
	
	You could modify the code to give the "rating" property a default value of 0:
	
	`let $rating := 0 ! xs:float(.)`
	
	Alternatively, if an XML source envelope contains the desired value in its attachments, you could extract it as follows:
	
	`let $rating := $source-node/rating ! xs:float(.)=>`
	`map:with('rating', xs:float(`
	`       $source//es:attachments/Person/rating/fn:data()))`

Use Case	Notes on the Generated Code

Property in Source Only	The default code contains only a commented out line you can use as a basic template for extraction, if appropriate. If this property has no analog in the target model, you can remove or ignore the commented out code.
	
	For example, if only the source contains a property named “address”, then the generated code includes the following:
	
	`(: The following properties are in the source, but not the target`
	`=> map:with('NO TARGET',`
	`xs:string($source-node/Person/address))`
	`:)`

Rename a Property	This appears as if the property in the source model was removed and a new property was added to the target model. Treat it like the “Property in Target Only” case, above, but use the original property as the source value.
	
	For example, if the source model contains a property named “firstName” that you change to “first”, then the default code contains the following:
	
	`let $first := $source-node/first ! xs:string(.)`
	
	Modify it to pull the value from the “firstName” property of the source:
	
	`let $first := $source-node/`**`firstName`**` ! xs:string(.)`

Entity Type Added to Target Model	A conversion function is generated that copies properties from the input source node to the output instance as if there are no differences. The code is equivalent to what `es:instance-converter-generate` produces. You should usually customize this function. For example, you could modify it to extract the new entity type property values from the raw source attachment of a source envelope. You could also use raw source as input to this function, rather than an envelope document.
Entity Type Removed from Source Model	A commented out conversion function is generated that copies properties from the input source node to the output instance as if there are no differences. You must uncomment and customize this function if you plan to store the values from instances of the defunct entity type somewhere in instances based on the target model.

4.5 Generating a TDE Template

You can generate a Template Driven Extraction (TDE) template from your model using Entity Services. Once installed, the template enables the following capabilities for your model-based application:

- Query your entity instances as row data using SQL or the Optic API.

- Query facts about and infer connections between your entity instances using SPARQL or the Optic API.

  Note: You can only take advantage of these capabilities for entity types that define a primary key. Without a primary key, there is no way to uniquely identify entity instances. For details on defining a primary key, see "Identifying the Primary Key Entity Property" on page 67.

This section contains the following topics:

To learn more about TDE, see Template Driven Extraction (TDE) in the *Application Developer's Guide*.

4.5.1 Generating a TDE Template

Use the `es:extraction-template-generate` XQuery function or the `es.extractionTemplateGenerate` JavaScript function to create a TDE template. The input to the template generation function is a JSON or `json:object` representation of a model descriptor. You can use the template as-is, or customize it for your application. You must install the template before your application can benefit from it. For details, see "Deploying a TDE Template" on page 126.

Note: Any hyphens ("-") in the model title, entity type names, or entity property names are converted to underscores ("_") when used in the generated template, in order to avoid invalid SQL names.

For example, the following code snippet generates a template from a model previously persisted in the database. For a more complete example, see "Example: TDE Template Generation and Deployment" on page 127.

Language	Example
XQuery	`es:extraction-template-generate(` `  fn:doc('/es-gs/models/person-1.0.0.json'))`
JavaScript	`es.extractionTemplateGenerate(` `  cts.doc('/es-gs/models/person-1.0.0.json'));`

The template is an important project artifact that you should put under source contol.

If you customize the template, you should validate it. You can use the `tde:validate` XQuery function or the `tde.validate` JavaScript function for standalone validation, or combine validation with insertion, as described in "Deploying a TDE Template" on page 126.

4.5.2 Characteristics of a Generated Template

A TDE template generated by the Entity Services API is intended to apply to entity envelope documents with the structure produced by an instance converter module. If you use a different structure, you will have to customize the template. For more details, see "What is an Envelope Document?" on page 150.

The generated template has the following characteristics:

- The default root context for the template matches instance data in both XML and JSON envelopes, assuming the envelopes conform to the Entity Services envelope convention. The generated template includes comments on how to change the context path for better performance if you only use a single envelope format (only XML or only JSON).
- A triples sub-template is defined for each entity type in the model that defines a primary key. This enables Semantic queries and inferencing on entity instances. For details, see "Triples Sub-Template Characteristics" on page 124.
- A rows sub-template is defined for each entity type in the model that defines at least one required property. This enables querying instances as rows using SQL or the Optic API. For details, see "Rows Sub-Template Characteristics" on page 125 and "Rows Template Array Property View Characteristics" on page 125.
- If you define a namespace prefix for an entity type as described in "Defining a Namespace URI for an Entity Type" on page 71, the prefix is used in XPath expressions in the template. Namespace prefixes are not used for references to entity types external to the model because such prefixes are unknown to the template generator.

4.5.2.1 Triples Sub-Template Characteristics

The triples sub-template for an entity type *T* has the following characteristics.

- A triples sub-template is only generated for entity types that define a primary key.
- The context for the sub-template is `./T`. That is, `//es:instance/T` in an envelope document. For example, `//es:instance/Person` if the model defines a `Person` entity type.
- A subject identifier variable named `subject-iri` is defined. The value of this variable is an IRI created by concatenating the entity type name with an instance's primary key value. This IRI identifies a particular instance of the entity type.
- A `triples` specification that will cause the following facts (triples) to be generated about each instance of type *T*:

- "This entity has type *T*", where the entity is identified by its primary key, and the type is identified by the `subject-iri` of the entity type. In RDF terms, the triple expresses "`<subject-iri> a <entity-type-iri>`".
- "This entity is defined by this model", where the entity is identified by its primary key, and the model is identified by the persisted descriptor URI. In RDF terms, the triple expresses "`<subject-iri> rdfs:isDefinedBy <descriptor-document-uri>`". This triple defines how to join instance/class membership to the instance document.

4.5.2.2 Rows Sub-Template Characteristics

The rows sub-template for an entity type *T* has the following characteristics.

- A rows sub-template is only generated for entity types that define at least one required property. (A primary key property is implicitly a required property.)
- The context for the sub-template is `./T`. That is, `//es:instance/T` in an envelope document.
- The schema name for the sub-template is the same as model title.
- For each entity property that does not have array type, a column with same name as the property is defined. (A property with array type is supported with a related view, so it is not present in the main view.)
- For each entity property with array type, a separate view named `T_propertyName` is defined. For example, `Person_friends`, if the `Person` entity type has an array typed property named `friends`. The characteristics of this view are described below.
- An entity property with `iri` as its data type is indexed as `IRI`.
- Any entity property that is not required is marked as nullable.

4.5.2.3 Rows Template Array Property View Characteristics

The `T_propertyName` view generated in the rows sub-template for an entity property with array type has the following characteristics:

- If the array item type is a scalar type, the view has two colums:
  - The left column has the same name and type as the primary key of the enclosing entity type (`T`).
  - The right column contains the scalar values in the array, each in its own row.
- If the array item type is a local reference and the referenced type defines a primary key, then view has two columns:
  - The left column has the same name and type as the primary key of the enclosing entity type (`T`).

    - The right column has the name *arrayPropName_primaryKey* and contains the primary key of the referenced type.
- If the array item type is a local reference and the referenced type does not define a primary key, then:
    - The leftmost column of the view has the same name and type as the primary key of the enclosing entity type (*T*).
    - There is a column for each property of the referenced type.
- If the array item type is an external reference, then the view has two columns:
    - The left column of the view has the same name and type as the primary key of the enclosing entity type (*T*).
    - The right column has the same name as the array property and type string. You usually need to customize this column definition.

4.5.3 Customizing a TDE Template

The following entity type characteristics result in a TDE template that requires customization:

- If no primary key is defined for an entity type that contains an array-typed property, you will like need to customize the template to define an appropriate type and value for the left column in the array view. This view is discussed in more detail in “Rows Template Array Property View Characteristics” on page 125.
- The template generator cannot determine the data type of an external entity type reference, so it defaults to string. You must manually set the type in the template.
- If you choose to embed entity instances inside one another, then the `context` element of the embedded type must be changed to reflect its position in instance documents.

You can make other customizations required by your application. For example, you might want to generate additional facts about your instances, or remove some columns from a row sub-template.

The generated template should work for both XML and JSON envelope documents in most cases, but some entity type structures might require customization of XPath expressions in the template in order to accomodate both formats.

For more details on the structure and content of TDE templates, see Template Driven Extraction (TDE) in the *Application Developer's Guide*.

4.5.4 Deploying a TDE Template

You must install your TDE template in the schemas database associated with your content database. The template must be in the special collection `http://marklogic.com/xdmp/tde` for MarkLogic to recognize it as template document.

Choose one of the following template installation methods:

- Use the `tde:template-insert` XQuery function or the `tde.templateInsert` JavaScript function. This method combines validation and installation in one step, and automatically inserts the template into the required collection.
- Use any general-purpose document insertion interface, such as `xdmp:document-insert` (XQuery) or `xdmp.documentInsert` (JavaScript). You must explicitly insert the template document into the special collection `http://marklogic.com/xdmp/tde`. No validation is performed.

For more details, see Validating and Inserting a Template in the *Application Developer's Guide*.

Once your template is installed, MarkLogic will update the row index and generate triples related to your instances whenever you ingest instances or reindexing occurs.

4.5.5 Example: TDE Template Generation and Deployment

The following example generates a TDE template from the model used in "Getting Started With Entity Services" on page 19, and then installs the template in the schemas database.

The following code generates a template from a previously persisted model, and then saves the template to a file on the filesystem as *$ARTIFACT_DIR*`/person-templ.xml`.

Language	Example

XQuery	`xquery version "1.0-ml";`
	`import module namespace es =`
	`  "http://marklogic.com/entity-services"`
	`  at "/MarkLogic/entity-services/entity-services.xqy";`
	
	`let $ARTIFACT_DIR := '/space/es/gs/'`
	`return xdmp:save(`
	`  fn:concat($ARTIFACT_DIR, 'person-templ.xml'),`
	`  es:extraction-template-generate(`
	`  fn:doc('/es-gs/models/person-1.0.0.json')))`

JavaScript	`'use strict';`
	`const es = require('/MarkLogic/entity-services/entity-services');`
	
	`const ARTIFACT_DIR = '/space/es/gs/';`
	`xdmp.save(`
	`  ARTIFACT_DIR + 'person-templ.xml',`
	`  es.extractionTemplateGenerate(`
	`  cts.doc('/es-gs/models/person-1.0.0.json'))`
	`);`

You are not required to save the template to the filesystem. However, the template is an important project artifact that you should place under source control. Saving the template to the filesystem makes it easier to do so.

If you apply the code above to the model from “Getting Started With Entity Services” on page 19, the resulting template defines two sub-templates. The first sub-template defines how to extract semantic triples from `Person` entity instances. The second sub-template defines how to extract a row-oriented projection of `Person` entity instances.

```
<template xmlns="http://marklogic.com/xdmp/tde">
...
  <templates>
    <template xmlns:tde="http://marklogic.com/xdmp/tde">
      <context>./Person</context>
      <vars>
        <var>
          <name>subject-iri</name>
          <val>sem:iri(...)</val>
        </var>
        ...
      </vars>
      <triples>...</triples>
    </template>
    <template xmlns:tde="http://marklogic.com/xdmp/tde">
      <context>./Person</context>
      <rows>...</rows>
      ...
    </template>
  </templates>
</template>
```

If the model includes additional entity types, then the template contains additional, similar sub-templates for these types.

The following code validates and installs a template using the convenience function provided by the TDE library module. Evaluate this code in the context of your content database.

Language	Example

XQuery	`xquery version "1.0-ml";`
	`import module namespace tde = "http://marklogic.com/xdmp/tde"`
	`  at "/MarkLogic/tde.xqy";`
	
	`let $ARTIFACT_DIR := '/space/es/gs/'`
	`return tde:template-insert(`
	`  '/es-gs/templates/person-1.0.0.xml',`
	`  xdmp:document-get(`
	`  fn:concat($ARTIFACT_DIR, 'person-templ.xml'))`
	`)`

JavaScript	`'use strict';`
	`const tde = require('/MarkLogic/tde');`
	
	`const ARTIFACT_DIR = '/space/es/gs/';`
	`tde.templateInsert(`
	`  '/es-gs/templates/person-1.0.0.xml',`
	`  fn.head(xdmp.documentGet(ARTIFACT_DIR + 'person-templ.xml'))`
	`);`

If the query runs sucessfully, the document `/es-gs/templates/person-1.0.0.xml` is created in the schemas database. If you explore the schemas database in Query Console, you should see that the template is in the special collection `http://marklogic.com/xdmp/tde`.

4.6 Generating an Entity Instance Schema

Entity Services can generate an XSD schema that you can use to validate canonical (XML) entity instances. Instance validation can be especially useful if you have a client or middle tier application submitting instances.

This section contains the following topics:

- Schema Generation Overview
- Schema Characteristics
- Schema Customization
- Example: Generating and Installing an Instance Schema
- Example: Validating an Instance Against a Schema

4.6.1 Schema Generation Overview

To generate a schema, apply the `es:schema-generate` XQuery function or the `es.schemaGenerate` JavaScript function to the `object-node` or `json:object` representation of a model descriptor, as shown in the following table. For a more complete example, see "Example: Generating and Installing an Instance Schema" on page 131.

Language	Example
XQuery	`es:schema-generate(fn:doc('/es-gs/models/person-1.0.0.json'))`
JavaScript	`es.schemaGenerate(cts.doc('/es-gs/models/People-1.0.0.json'));`

The schema is an important project artifact, so you should place it under source control.

Before you can use the generated schema(s) for instance validation, you must deploy the schema to the schemas database associated with your content database. You can use any of the usual document insertion APIs for this operation.

> **Note:** If your model defines multiple entity types and the entity type definitions do not all use the same namespace, a schema is generated for each unique namespace. Install all of the generated schemas in the schemas database.

Use the `xdmp:validate` XQuery function or the `xdmp.validate` JavaScript function to validate instances against your schema. For an example, see "Example: Validating an Instance Against a Schema" on page 133.

Note that you can only validate entity instances expressed as XML. You can extract the XML representation of an instance from an envelope document using the `es:instance-xml-from-document` XQuery function or the `es.instanceXmlFromDocument` JavaScript function.

4.6.2 Schema Characteristics

The Entity Services API applies the following rules when generating a schema from a model:

- A scalar property type is translated into a simple, type-enforced `xs:element`.
- The schema includes an `xs:complexType` for each entity type defined by the model. This type contains a sequence of elements representing the entity type properties.
- For each external entity type reference, a type is generated that can hold a value for a reference of that type by using the string after the last slash ('/') in the external reference URI.
- For each local entity type reference, an `es:complexType` is generated.

- Array typed entity properties are handled using `minOccurs` and `maxOccurs` on the property's `xs:element`.
- Any entity property that is not a primary key or required is set to `minOccurs="0"`.
- A required property has cardinality 1.
- The automated schema generation cannot resolve multiple properties with same name, but different data type. If this occurs, an `xs:element` is generated for one property, and then the `xs:element` definitions for the other properties will be commented out. You must customize the schema (or modify your model) to resolve this conflict.
- A separate schema is generated for each namespace declared in the model. For more details on using namespaces in entity type definitions, see "Defining a Namespace URI for an Entity Type" on page 71.

4.6.3 Schema Customization

The following list describes some situations in which schema customization might be needed.

- If your model contains multiple entity type properties with the same name, only one of them will be reflected in the schema. The other(s) will be commented out. Change the schema (or your model) to resolve this conflict.
- Depending on how entity references are used in the model, parts of the schema might be superfluous and can be removed.
- You might have to choose between validating entity references or validating embedded entity instances, depending on the choices you make with respect to normalization and entity document structure.

4.6.4 Example: Generating and Installing an Instance Schema

The following example generates a schema from a previously persisted model, and then inserts it into the schemas database.

Since the model is in the content database and the schema must be inserted into the schemas database, `xdmp:eval` is used to switch database contexts for the schema insertion. If you generated the schema and saved it to the filesystem first, then you would only have to work with the schemas database, so the eval would be unnecessary.

The following code inserts a schema with the URI `/es-gs/person-1.0.0.xsd` into the schemas database associated with the content database that holds the source model. Assume the model was previously persisted as a document with URI `/es-gs/models/person-1.0.0.json`.

Language	Example

XQuery	`xquery version "1.0-ml";`
	`import module namespace es =`
	`"http://marklogic.com/entity-services"`
	`  at "/MarkLogic/entity-services/entity-services.xqy";`
	
	`(: The query that inserts the schema into the schemas db :)`
	`let $query :=`
	`  'xquery version "1.0-ml";`
	`  declare variable $schema as element(xs:schema) external;`
	`  declare variable $uri as xs:string external;`
	`  xdmp:document-insert($uri, $schema)'`
	
	`(: Generate the schema :)`
	`let $schema :=`
	`  **es:schema-generate**(fn:doc('/es-gs/models/person-1.0.0.json'))`
	
	`(: Insert the schema into the Schemas db :)`
	`return xdmp:eval($query,`
	`  (xs:QName("schema"), $schema,`
	`   xs:QName("uri"), '/es-gs/person-1.0.0.xsd'),`
	`  <options xmlns="xdmp:eval">`
	`    <database>{xdmp:schema-database()}</database>`
	`  </options>`
	`)`

JavaScript	`'use strict';`
	`const es = require('/MarkLogic/entity-services/entity-services')`
	
	`// The query that inserts the schema into the schemas db`
	`const query = 'declareUpdate(); xdmp.documentInsert(uri,`
	`schema);'`
	`// Generate the schema`
	`const schema = fn.head(`
	`  **es.schemaGenerate**(cts.doc('/es-gs/models/person-1.0.0.json')));`
	`xdmp.eval(`
	`  query,`
	`  {schema: schema, uri: '/es-gs/person-1.0.0.xsd'},  // vars`
	`  {database: xdmp.schemaDatabase()}  // options`
	`);`

4.6.5 Example: Validating an Instance Against a Schema

The following example validates an instance against a schema generated using the `es:schema-generate` XQuery function or the `es.schemaGenerate` Server-Side JavaScript function. It is assumed that the schema is already installed in the schema database associated with the content database, as shown in "Example: Generating and Installing an Instance Schema" on page 131.

The following code validates an entity instance within a previously persisted envelope document. Assume this instance was created using the instance converter module for its entity type, and therefore is valid. Thus, the validation succeeds. The query returns an empty `xdmp:validation-errors` element in XQuery and an empty object in JavaScript.

Language	Example

XQuery	`xquery version "1.0-ml";`
	`import module namespace es =`
	`  "http://marklogic.com/entity-services"`
	`  at "/MarkLogic/entity-services/entity-services.xqy";`
	
	`xdmp:validate(`
	`  es:instance-xml-from-document(`
	`  fn:doc('/es-gs/envelopes/1234.xml')),`
	`  'type', xs:QName('PersonType'))`

JavaScript	`'use strict';`
	`const es = require('/MarkLogic/entity-services/entity-services')`
	
	`xdmp.validate(`
	`  es.instanceXmlFromDocument(`
	`  cts.doc('/es-gs/envelopes/1234.xml')),`
	`  'type', 'PersonType')`

The following example validates an in-memory instance against the schema. The schema is based on the model from “Getting Started With Entity Services” on page 19. The instance was intentionally created without a required property (“id”) so that it will fail validation.

Language	Example

XQuery	`xquery version "1.0-ml";`
	`import module namespace es =`
	`"http://marklogic.com/entity-services"`
	`  at "/MarkLogic/entity-services/entity-services.xqy";`
	
	`let $invalid-entity :=`
	`  <Person>`
	`  <firstName>George</firstName>`
	`  <lastName>Washington</lastName>`
	`  <fullName>George Washington</fullName>`
	`  </Person>`
	`return`
	`xdmp:validate($invalid-entity, 'type', xs:QName('PersonType'))`

JavaScript	`'use strict';`
	`const invalidEntity = fn.head(xdmp.unquote(`
	`  '<Person>'+`
	`  '<firstName>George</firstName>' +`
	`  '<lastName>Washington</lastName>' +`
	`  '<fullName>George Washington</fullName>' +`
	`'</Person>'));`
	`xdmp.validate(invalidEntity, 'type', 'PersonType');`

4.7 Generating a PII Security Configuration Artifact

You identify PII entity properties using the `pii` property of an entity model, as described in “Identifying Personally Identifiable Information (PII)” on page 69. Then, use the `es:pii-generate` XQuery function or the `es.piiGenerate` JavaScript function to generate a security configuration artifact that enables stricter access control for PII entity instance properties.

The generated configuration contains an Element Level Security (ELS) protected path definition for each PII property, and an ELS query roleset configuration. The protected path configuration limits read access to users with the “pii-reader” security role. The query roleset prevents users without the “pii-reader” role from seeing the protected content in response to a query or XPath expression. The “pii-reader” role is pre-defined by MarkLogic.

To learn more about Element Level Security, protected paths, and query rolesets, see Element Level Security in the *Security Guide*.

For example, the following model descriptors specify that the `name` and `bio` properties can contain PII:

Format	Example Model Descriptor

JSON	`{ "info": {`
	`  "title": "People",`
	`  "description": "People Example",`
	`  "version": "4.0.0"`
	`},`
	`"definitions": {`
	`  "Person": {`
	`  "properties": {`
	`  "id": { "datatype": "int" },`
	`  "name": { "datatype": "string" },`
	`  "bio": { "datatype": "string" },`
	`  "rating": { "datatype": "float" }`
	`  },`
	`  "required": ["name"],`
	`  "primaryKey": "id",`
	`  "pii": ["name", "bio"]`
	`}}}`

XML	`<es:model xmlns:es="http://marklogic.com/entity-services">`
	`  <es:info>`
	`  <es:title>People</es:title>`
	`  <es:description>People Example</es:description>`
	`  <es:version>4.0.0</es:version>`
	`  </es:info>`
	`  <es:definitions>`
	`  <Person>`
	`  <es:properties>`
	`  <id><es:datatype>int</es:datatype></id>`
	`  <name><es:datatype>string</es:datatype></name>`
	`  <bio><es:datatype>string</es:datatype></bio>`
	`  <rating><es:datatype>float</es:datatype></rating>`
	`  </es:properties>`
	`  <es:required>name</es:required>`
	`  <es:primary-key>id</es:primary-key>`
	`  <es:pii>name</es:pii>`
	`  <es:pii>bio</es:pii>`
	`  </Person>`
	`  </es:definitions>`
	`</es:model>`

Assuming the above model descriptor is persisted in the database as `/es-ex/models/people-4.0.0.json`, then the following code generates a database configuration artifact from the model:

Language	Example

XQuery	`xquery version "1.0-ml";`
	`import module namespace es =`
	`"http://marklogic.com/entity-services"`
	`at "/MarkLogic/entity-services/entity-services.xqy";`
	
	`es:pii-generate(`
	`fn:doc('/es-ex/models/people-4.0.0.json'))`

JavaScript	`'use strict';`
	`const es = require('/MarkLogic/entity-services/entity-services');`
	
	`es.piiGenerate(`
	`cts.doc('/es-ex/models/people-4.0.0.json'))`

The generated security configuration artifact should look similar to the following. If you deploy this configuration, then only users with the “pii-reader” security role can read the “name” and “bio” properties of a Person instance. The “pii-reader” role is pre-defined by MarkLogic.

```
{ "name": "People-4.0.0",
  "desc": "A policy that secures name,bio of type Person",
  "config": {
    "protected-path": [
      {
        "path-expression": "/envelope//instance//Person/name",
        "path-namespace": [],
        "permission": {
          "role-name": "pii-reader",
          "capability": "read"
        }
      },
      {
        "path-expression": "/envelope//instance//Person/bio",
        "path-namespace": [],
        "permission": {
          "role-name": "pii-reader",
          "capability": "read"
        }
      }
    ],
    "query-roleset": {
      "role-name": [
        "pii-reader"
      ]
} } }
```

Note that the configuration only includes protected paths for PII properties in the entity instance. Envelope documents also contain the original source document as an attachment by default. Any PII in the source attachment is not protected by the generated configuration. You might want to define additional protected paths or modify the `extract-instance-`*`T`* function of your instance converter module to exclude the source attachment.

Deploy the artifact using the Configuration Management API. For example, if the file `pii-config.json` contains the configuration generated by the previous example, then the following command adds the protected paths and query roleset to MarkLogic's security configuration:

```
curl --anyauth --user user:password -X PUT -i \
  -d @./pii-config.json -H "Content-type: application/json" \
  http://localhost:8002/manage/v3
```

You can add additional configuration settings to the generated artifact, or merge the generated settings into configuration settings created and maintained elsewhere. For example, you could configure additional protected paths to control access to the source data for the "name" and "bio" properties in the source attachment of your instance envelope documents.

4.8 Generating a Database Configuration Artifact

Use the `es:database-properties-generate` XQuery function or the `es.databasePropertiesGenerate` JavaScript function to create a database configuration artifact from the JSON `object-node` or `json:object` representation of a model descriptor. This artifact is helpful for configuring your content database. You are not required to use this artifact; it is a convenience feature.

The generated configuration information always has at least the following items, and may contain additional property definitions, depending on the model:

- Enable the triple index and the collection lexicon, both of which are required for querying a model as described in "Search Basics for Models" on page 170.
- Define the "es" namespace prefix globally so that it can be used in path queries.

If an entity type definition specifies entity properties for range index and word lexicon configuration, then the database configuration artifact includes corresponding index and/or lexicon configuration information.

For example, the following model descriptors specify a path range index for the `id` and `rating` properties and a word lexicon for the `bio` property of the `Person` entity type:

Format	Example Model Descriptor

JSON	`{ "info": {`
	`  "title": "People",`
	`  "description": "People Example",`
	`  "version": "3.0.0"`
	`},`
	`"definitions": {`
	`  "Person": {`
	`  "properties": {`
	`  "id": { "datatype": "int" },`
	`  "name": { "datatype": "string" },`
	`  "bio": { "datatype": "string" },`
	`  "rating": { "datatype": "float" }`
	`  },`
	`  "required": ["name"],`
	`  "primaryKey": "id",`
	`  **"pathRangeIndex": ["id", "rating"],**`
	`  **"wordLexicon": ["bio"]**`
	`}}}`

XML	`<es:model xmlns:es="http://marklogic.com/entity-services">`
	`  <es:info>`
	`  <es:title>People</es:title>`
	`  <es:description>People Example</es:description>`
	`  <es:version>3.0.0</es:version>`
	`  </es:info>`
	`  <es:definitions>`
	`  <Person>`
	`  <es:properties>`
	`  <id><es:datatype>int</es:datatype></id>`
	`  <name><es:datatype>string</es:datatype></name>`
	`  <bio><es:datatype>string</es:datatype></bio>`
	`  <rating><es:datatype>float</es:datatype></rating>`
	`  </es:properties>`
	`  <es:required>name</es:required>`
	`  <es:primary-key>id</es:primary-key>`
	`  **<es:path-range-index>id</es:path-range-index>**`
	`  **<es:path-range-index>rating</es:path-range-index>**`
	`  **<es:word-lexicon>bio</es:word-lexicon>**`
	`  </Person>`
	`  </es:definitions>`
	`</es:model>`

Assuming the above model descriptor is persisted in the database as `/es-ex/models/people-3.0.0.json`, then the following code generates a database configuration artifact from the model:

Language	Example

XQuery	`xquery version "1.0-ml";`
	`import module namespace es =`
	`  "http://marklogic.com/entity-services"`
	`  at "/MarkLogic/entity-services/entity-services.xqy";`
	
	`es:database-properties-generate(`
	`  fn:doc('/es-ex/models/people-3.0.0.json'))`

JavaScript	`'use strict';`
	`const es = require('/MarkLogic/entity-services/entity-services');`
	
	`es.databasePropertiesGenerate(`
	`  cts.doc('/es-ex/models/people-3.0.0.json'))`

The generated configuration artifact should look similar to the following. Notice that range index information is included for `id` and `rating` and word lexicon information is included for `bio`.

```
{
  "database-name": "%%DATABASE%%",
  "schema-database": "%%SCHEMAS_DATABASE%%",
  "path-namespace": [
    {
      "prefix": "es",
      "namespace-uri": "http://marklogic.com/entity-services"
    }
  ],
  "element-word-lexicon": [
    {
      "collation": "http://marklogic.com/collation/en",
      "localname": "bio",
      "namespace-uri": ""
    }
  ],
  "range-path-index": [
    {
      "collation": "http://marklogic.com/collation/en",
      "invalid-values": "reject",
      "path-expression": "//es:instance/Person/id",
      "range-value-positions": false,
      "scalar-type": "int"
    },
    {
      "collation": "http://marklogic.com/collation/en",
      "invalid-values": "reject",
      "path-expression": "//es:instance/Person/rating",
```

```
        "range-value-positions": false,
        "scalar-type": "float"
      }
    ],
    "triple-index": true,
    "collection-lexicon": true
}
```

Note that the generated range index configuration disables range value positions and rejects invalid values by default. You might choose to change one or both of these settings, depending on your application.

You can add additional configuration settings to the generated artifact, or merge the generated settings into configuration settings created and maintained elsewhere.

You can use the generated configuration properties with your choice of configuration interface. For example, you can use the artifact with the REST Management API (after minor modification), or you can extract the configuration information to use with the XQuery Admin API.

To use the generated database configuration artifact with the REST Management API method `PUT:/manage/v2/databases/{id|name}/properties`, make the following modifications:

- Replace `%%DATABASE%%` with the name of your content database.
- Replace `%%SCHEMAS_DATABASE%%` with the name of the schemas database associated with your content database.
- If you have configured other range indexes or word lexicons into your database, merge your existing index or lexicon configuration with the generated configuration so that no settings are lost.

For example, you can use a curl command similar to the following to change the properties of the database named “es-ex”. Assume the file `db-props.json` contains the previously shown config artifact above, with the `database-name` and `schema-database` property values modified to “es-ex” and “Schemas”, respectively.

```
curl --anyauth --user user:password -X PUT -i \
  -d @./db-props.json -H "Content-type: application/json" \
  http://localhost:8002/manage/v2/databases/es-ex/properties
```

If you then examine the configuration for the “es-ex” database using the Admin Interface or the REST Management API method `GET:/manage/v2/databases/{id|name}/properties`, you should see the expected range indexes and word lexicon have been created.

For more information about database configuration, see the following:

- `PUT:/manage/v2/databases/{id|name}/properties`
- Range Indexes and Lexicons in the *Administrator’s Guide*

- Using the Management API in the *Monitoring MarkLogic Guide*

4.9 Generating Query Options for Searching Instances

This section describes how to use the Entity Services API to generate a set of query options you can use to search entity instances using the XQuery Search API or the REST, Java, and Node.js Client APIs. This section covers the following topics:

- Options Generation Overview
- Characteristics of the Generated Options
- Example: Generating Query Options

For more details and examples, see "Querying a Model or Entity Instances" on page 169.

4.9.1 Options Generation Overview

Generate model-based query options using the `es:search-options-generate` XQuery function or the `es.searchOptionsGenerate` JavaScript function. Pass in the JSON `object-node` or `json:object` representation of a model descriptor.

For example, if the document `/es-gs/models/person-1.0.0.json` is a previously persisted descriptor, then you can generate query options from the model with one of the following calls.

Language	Example
XQuery	`es:search-options-generate(` `  fn:doc('/es-gs/models/person-1.0.0.json'))`
JavaScript	`es.searchOptionsGenerate(` `  cts.doc('/es-gs/models/person-1.0.0.json'));`

For a more complete example, see "Example: Generating Query Options" on page 144.

You can use the generated options in the following ways:

- Pass them as the second parameter of the `search:search` or `search:resolve` XQuery functions, or the `search.search` or `search.resolve` JavaScript functions.
- Embed them in a combined query used with the REST, Java, or Node.js APIs.
- Install them in the database and use them as persistent query options with the REST, Java, or Node.js APIs.
- Use them as a jumping off point for creating constraint bindings for use with the `cts:parse` XQuery function or the `cts.parse` JavaScript function. Then use the resulting cts:query object with `cts:search` or the JSearch API.

For an example and discussion of the options, see “Example: Using the Search API for Instance Queries” on page 176.

4.9.2 Characteristics of the Generated Options

The generated options include the following:

- A value constraint named “entity-type” for constraining searches to entities of a particular type. For example:

```
<search:constraint name="entity-type">
  <search:value>
    <search:element ns="http://marklogic.com/entity-services" name="title"/>
  </search:value>
</search:constraint>
```

- A URI value constraint named “uris”. For example:

```
<search:values name="uris">
  <search:uri/>
</search:values>
```

- An `extract-document-data` option for returning just the canonical entity instance(s) from matched documents. For example, the following option extracts just the `Person` entity instance from matched documents:

```
<search:extract-document-data selected="include">
  <search:extract-path xmlns:es="...">
    //es:instance/(Person)
  </search:extract-path>
</search:extract-document-data>
```

- An `additional-query` option that constrains results to documents containing `es:instance` elements. For example:

```
<search:additional-query>
  <cts:element-query xmlns:cts="http://marklogic.com/cts">
    <cts:element xmlns:es="...">es:instance</cts:element>
    <cts:true-query/>
  </cts:element-query>
</search:additional-query>
```

- Options that disable faceting and snippeting (in favor of just extracting the instances). For example:

```
<search:return-facets>false</search:return-facets>
<search:transform-results apply="empty-snippet"/>
```

- An option that enables unfiltered search. For example:

```
<search:search-option>unfiltered</search:search-option>
```

- If the model defines a primary key, a value constraint on the primary key property. For example:

```
<search:constraint name="id">
  <search:value>
    <search:element ns="" name="id"/>
  </search:value>
</search:constraint>
```

- For each property named in the `pathRangeIndex` or `rangeIndex` property of an entity type definition, a path range index constraint with the same name as the entity property. For example:

```
<search:constraint name="rating">
  <search:range type="xs:float" facet="true">
    <search:element ns="" name="rating" />
  </search:range>
</search:constraint>
```

- For each property named in the `elementRangeIndex` property of an entity type definition, an element range index constraint with the same name as the entity property. For example:

```
<search:constraint name="rating">
  <search:range type="xs:float" facet="true">
    <search:path-index xmlns:es="...">
      //es:instance/Person/rating
    </search:path-index>
  </search:range>
</search:constraint>
```

- For each property named in the `wordLexicon` property of an entity type definition, a word constraint with the same name as the entity property. For example:

```
<search:constraint name="bio">
  <search:word>
    <search:element ns="" name="bio"/>
  </search:word>
</search:constraint>
```

- If an entity type includes more than one property in the range index specification, a `tuples` option with the same name as the entity type for finding co-occurrences of the indexed properties. For example:

```
<search:tuples name="Item">
  <search:range type="xs:int" facet="true">
    <search:path-index xmlns:es="...">
      //es:instance/Item/price
    </search:path-index>
  </search:range>
  <search:range type="xs:float" facet="true">
    <search:path-index xmlns:es="...">
      //es:instance/Item/rating
```

```
      </search:path-index>
    </search:range>
  </search:tuples>
```

The generated options include extensive comments to assist you with customization. The options are usable as-is, but optimal search configuration is highly application dependent, so it is likely that you will extend or modify the generated options.

If the primary key property is also listed in the range index specification, then both a value constraint and a range constraint would be generated with the same name. Since this is not allowed, one of these constraints will be commented out. You can change the name and uncomment it. For an example of this conflict, see “Example: Generating Query Options” on page 144.

4.9.3 Example: Generating Query Options

The following example generates a set of query options from a model and saves the results to a file on the filesystem so you can place it under source control or make modifications.

This example assumes the following descriptor has been inserted into the database with the URI `/es-ex/models/people-1.0.0.json`.

```
{ "info": {
    "title": "People",
    "description": "People Example",
    "version": "1.0.0"
  },
  "definitions": {
    "Person": {
      "properties": {
        "id": { "datatype": "int" },
        "name": { "datatype": "string" },
        "bio": { "datatype": "string" },
        "rating": { "datatype": "float" }
      },
      "required": [ "name" ],
      "primaryKey": "id",
      "pathRangeIndex": [ "id", "rating" ],
      "wordLexicon": [ "bio" ]
}}}
```

The following code generates a set of query options from the above model. The options are saved to the file *ARTIFACT_DIR*/people-options.xml.

Language	Example

XQuery	`xquery version "1.0-ml";`
	`import module namespace es =`
	`  "http://marklogic.com/entity-services"`
	`  at "/MarkLogic/entity-services/entity-services.xqy";`
	
	`let $ARTIFACT_DIR := '/space/es/ex/'  (: CHANGE THIS VALUE :)`
	`return xdmp:save(`
	`  fn:concat($ARTIFACT_DIR, 'people-options.xml'),`
	`  es:search-options-generate(`
	`  fn:doc('/es-ex/models/people-1.0.0.json')))`

JavaScript	`'use strict';`
	`const es = require('/MarkLogic/entity-services/entity-services');`
	
	`const ARTIFACT_DIR = '/space/es/ex/';  // CHANGE THIS VALUE`
	`xdmp.save(`
	`  ARTIFACT_DIR + 'people-options.xml',`
	`  es.searchOptionsGenerate(`
	`  cts.doc('/es-ex/models/people-1.0.0.json'))`
	`);`

The resulting options should be similar to the following.

```
<search:options
    xmlns:search="http://marklogic.com/appservices/search">
  <search:constraint name="entity-type">
    <search:value>
      <search:element ns="http://marklogic.com/entity-services"
name="title"/>
    </search:value>
  </search:constraint>
  <search:constraint name="id">
    <search:value>
      <search:element ns="" name="id"/>
    </search:value>
  </search:constraint>
  <!--This item is a duplicate and is commented out so as to create
      a valid artifact.
 <search:constraint name="id"
    xmlns:search="http://marklogic.com/appservices/search">
  <search:range type="xs:int" facet="true">
    <search:path-index
        xmlns:es="http://marklogic.com/entity-services">
      //es:instance/Person/id
    </search:path-index>
  </search:range>
```

```
</search:constraint>
-->
  <search:constraint name="rating">
    <search:range type="xs:float" facet="true">
      <search:path-index
          xmlns:es="http://marklogic.com/entity-services">
        //es:instance/Person/rating
      </search:path-index>
    </search:range>
  </search:constraint>
  <search:constraint name="bio">
    <search:word>
      <search:element ns="" name="bio"/>
    </search:word>
  </search:constraint>
  <search:tuples name="Person">
    <search:range type="xs:int" facet="true">
      <search:path-index
          xmlns:es="http://marklogic.com/entity-services">
        //es:instance/Person/id
      </search:path-index>
    </search:range>
    <search:range type="xs:float" facet="true">
      <search:path-index
          xmlns:es="http://marklogic.com/entity-services">
        //es:instance/Person/rating
      </search:path-index>
    </search:range>
  </search:tuples>
  <!--Uncomment to return no results for a blank search, rather
      than the default of all results
  <search:term xmlns:search="http://marklogic.com/appservices/search">
    <search:empty apply="no-results"/>
  </search:term>
-->
  <search:values name="uris">
    <search:uri/>
  </search:values>
  <!--Change to 'filtered' to exclude false-positives in certain
      searches-->
  <search:search-option>unfiltered</search:search-option>
  <!--Modify document extraction to change results returned-->
  <search:extract-document-data selected="include">
    <search:extract-path
        xmlns:es="http://marklogic.com/entity-services">
      //es:instance/(Person)
    </search:extract-path>
  </search:extract-document-data>
  <!--Change or remove this additional-query to broaden search
      beyond entity instance documents-->
  <search:additional-query>
    <cts:element-query xmlns:cts="http://marklogic.com/cts">
      <cts:element xmlns:es="http://marklogic.com/entity-services">
        es:instance
```

```
      </cts:element>
      <cts:true-query/>
    </cts:element-query>
  </search:additional-query>
  <!--To return facets, change this option to 'true' and edit
constraints-->
  <search:return-facets>false</search:return-facets>
  <!--To return snippets, comment out or remove this option-->
  <search:transform-results apply="empty-snippet"/>
</search:options>
```

Notice that two constraints are generated for the `id` property. A value constraint is generated because `id` is the primary key for a `Person` entity. A path range constraint is generated because `id` is listed in the `pathRangeIndex` property of the `Person` entity type definition. Since it is not possible for two constraints to have the same name in a set of options, the second constraint is commented out:

```
<search:constraint name="id">
    <search:value>
      <search:element ns="" name="id"/>
    </search:value>
  </search:constraint>
  <!--This item is a duplicate and is commented out so as to create
      a valid artifact.
 <search:constraint name="id"
    xmlns:search="http://marklogic.com/appservices/search">
  <search:range type="xs:int" facet="true">
    <search:path-index
        xmlns:es="http://marklogic.com/entity-services">
      //es:instance/Person/id
    </search:path-index>
  </search:range>
</search:constraint>
```

If you do not need both constraint types on id, you can remove one of them. Alternatively, you can change the name of at least one of these constraints and uncomment the path range constraint.

For an example of using the generated options, see “Example: Using the Search API for Instance Queries” on page 176.

4.10 Deploying Generated Code and Artifacts

Library modules and some configuration artifacts that you generate using the Entity Services API must be installed before you can use them.

- Code modules: Insert into the modules database associated with your App Server.

  For example, if you’re using the pre-configured App Server on port 8000, insert your instance converter module into the Modules database. For more details, see Importing XQuery Modules, XSLT Stylesheets, and Resolving Paths in the *Application Developer’s Guide*.

- Schemas: Insert into the schemas database associated with your content database.

  For example if your content database is the pre-configured Documents database, deploy schemas to the Schemas database.

- TDE templates: Insert into the schemas database associated with your content database.

  For example if your content database is the pre-configured Documents database, deploy templates to the Schemas database. For details, see “Deploying a TDE Template” on page 126.

- Database configuration: This artifact does not require installation. Rather, you use it as input during configuration operations, as described in “Generating a Database Configuration Artifact” on page 137.
- Query Options: Installation on MarkLogic is optional. If you choose to use these as persistent options with the Java, Node.js, or REST Client APIs, see “Pre-Installing Query Options” on page 173. Otherwise, no installation is required.

Unless otherwise noted, you can install a module or configuration artifact using any document insertion interfaces, including the following MarkLogic APIs:

- The `xdmp:document-insert` XQuery function or the `xdmp.documentInsert` Server-Side JavaScript function.
- The Java, Node.js, and REST Client APIs. The Client APIs include interfaces specifically for managing documents in the modules database associated with a REST API instance, as well as normal document operations that can be performed against any database.

For an example of deploying a module using simple document insert, see “Create and Deploy an Instance Converter” on page 27 (XQuery) or “Create and Deploy an Instance Converter” on page 43 (JavaScript).

In addition, open source application deployment tools such as `ml-gradle` and `roxy` (both available on GitHub) support module deployment tasks. The Entity Services examples on GitHub use `ml-gradle` for this purpose; for more details, see “Exploring the Entity Services Open-Source Examples” on page 15.

5.0 Managing Entity Instances

This chapter describes how to create, retrieve, update, and delete entity instances derived from a model created with MarkLogic Entity Services. The chapter covers the following topics:

5.1 Entity Instance Concepts

This section introduces entity instance concepts helpful in creating, persisting, querying, and extracting entity instance data. The following topics are included:

5.1.1 What is an Instance?

An entity instance is a concrete instantiation of an entity type defined in a model.

For example, suppose you have a JSON model descriptor that defines a `Person` entity type with the following properties. This is based on the model in “Getting Started With Entity Services” on page 19.

```
"Person": {
  "properties": {
    "id": {"datatype": "string"},
    "firstName": {"datatype": "string"},
    "lastName": {"datatype": "string"},
    "fullName": {"datatype": "string"},
    "friends": {
      "datatype": "array",
      "items": {"$ref": "#/definitions/Person"
    }
  }},
  ...
}
```

Then the canonical representation of a `Person` instance would have the following form, depending on whether you choose to work with XML or JSON.

XML Canonical Form	JSON Canonical Form

<Person>
  <id>1234</id>
  <firstName>George</firstName>
  <lastName>Washington</lastName>
  <fullName>George Washington</fullName>
</Person>
	{"Person": {
	  "id":"2345",
	  "firstName":"Martha",
	  "lastName":"Washington",
	  "fullName":"Martha Washington"
	}}

By convention, an instance is stored as child XML elements or JSON properties of an envelope document. You can extract an instance from an envelope as XML or JSON, regardless of the envelope format. For details, see “What is an Envelope Document?” on page 150 and “Extracting an Entity Instance from an Envelope Document” on page 161.

An instance can have multiple repesentations, depending on the context:

- While you are synthesizing an instance from raw source or converting one between model versions, you work with an in-memory representation of the instance as a `map:map` containing not only the entity type property values, but additional information such as type and source. This representation is designed to be easy to modify during instance construction.
- By Entity Services convention, instances are persisted in envelope documents. An XML envelope document includes an `es:instance` XML element with a child element that is the canonical XML representation of the instance. A JSON envelope document contains an `"instance"` property that contains the canonical JSON representation of the instance. The canonical representation is the one on which queries are based. For details, see “What is an Envelope Document?” on page 150.
- You can extract an instance from an envelope document as XML, JSON, or a `map:map`. You might use one or more of these representations to pass instances to downstream applications. For details, see “Extracting an Entity Instance from an Envelope Document” on page 161.

For more details, see “Example: Entity Instance Representations” on page 152.

5.1.2 What is an Envelope Document?

If you follow the Entity Services conventions, your entity instances are persisted in MarkLogic as part of an envelope document. An envelope document encapsulates instance data with related metadata that might be useful to your application. You can use either XML or JSON envelopes.

An envelope document for some entity type *T* is created using the `instance-to-envelope` function in *T*'s instance converter module. For more details, see "Creating an Entity Instance from a Data Source" on page 157 and "Creating an Instance Converter Module" on page 110.

An envelope document has the following form by default.

Format	Envelope Template

XML	`<es:envelope xmlns:es="http://marklogic.com/entity-services">`
	`  <es:instance>`
	`  <es:info>`
	`  <es:title>model title</es:title>`
	`  <es:version>model version</es:version>`
	`  </es:info>`
	`  <T>`
	`  ...T's entity properties as elements...`
	`  </T>`
	`  </es:instance>`
	`  <es:attachments>...source data...</es:attachments>`
	`</es:envelope>`

JSON	`{"envelope": {`
	`  "instance": {`
	`  "info": {`
	`  "title": "model title",`
	`  "version": "model version"`
	`  },`
	`  "T": {`
	`  ...T's entity properties as JSON properties...`
	`  }`
	`  },`
	`  "attachments": [ ...source data... ]`
	`}}`

The `instance` section contains the canonical representation of the instance, plus metadata such as the model title and version from which entity type is derived. The `attachments` section contains the source data, by convention; you can add additional attachments.

The envelope format does not have to match the format of your raw source data. You can generate JSON envelopes for instances based on XML source and vice versa. However, if the source and envelope formats differ, the raw source is stored in the `attachments` section of the envelope as a string.

You can customize an envelope document to include other information, but you should generally not modify the `instance` portion. The instance data should accurately reflect the entity type definition in your model. If you need to normalize or derive property values, do so in the `extract-instance-T` function of your instance converter.

If you customize the envelope by adding data to the `attachments` element, then you can use the `es:instance-get-attachments` XQuery function or the `es.instanceGetAttachments` JavaScript function to retrieve the data. If you put it elsewhere in the envelope, then you are solely responsible for retrieving it from the envelope.

The Entity Services API includes functions for retrieving the instance data and attachments from an envelope. For details, see “Extracting an Entity Instance from an Envelope Document” on page 161 and “Extracting the Original Source from an Envelope Document” on page 164.

5.1.3 Example: Entity Instance Representations

This example illustrates the various instance representations discussed in “What is an Instance?” on page 149.

- XML Entity Instance Representations
- JSON Entity Instance Representations

5.1.3.1 XML Entity Instance Representations

This example uses the `Person` entity type from the model defined in “Getting Started With Entity Services” on page 19.

	Representation	Example

1	Raw Source	`<person>`
		`  <pid>1234</pid>`
		`  <given>George</given>`
		`  <family>Washington</family>`
		`</person>`

2	In-memory instance, as returned by `extract-instance-Person`
	
	Shown here as JSON for readability, but really a `json:object` (`map:map`) with keys `$attachments`, `$type`, `id`, etc.
		`{"$attachments": "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n<person>\n <pid>1234</pid>\n  <given>George</given>\n <family>Washington</last>\n</family>",`
		`  "$type": "Person",`
		`  "id": "1234",`
		`  "firstName": "George",`
		`  "lastName": "Washington",`
		`  "fullName": "George Washington"`
		`}`

	Representation	Example

3	Canonical XML instance generated by `instance-to-canonical`
	
	Used to construct the instance within an envelope document.
		`<Person>`
		`  <id>1234</id>`
		`  <firstName>George</firstName>`
		`  <lastName>Washington</lastName>`
		`  <fullName>George Washington</fullName>`
		`</Person>`

4	Envelope document, as generated by `instance-to-envelope`	`<es:envelope`
		`  xmlns:es="http://marklogic.com/entity-services">`
		`  <es:instance>`
		`  <es:info>`
		`  <es:title>Person</es:title>`
		`  <es:version>1.0.0</es:version>`
		`  </es:info>`
		`  <Person>`
		`  <id>1234</id>`
		`  <firstName>George</firstName>`
		`  <lastName>Washington</lastName>`
		`  <fullName>George Washington</fullName>`
		`  </Person>`
		`  </es:instance>`
		`  <es:attachments>`
		`  <person>`
		`  <pid>1234</pid>`
		`  <first>George</first>`
		`  <last>Washington</last>`
		`  </person>`
		`  </es:attachments>`
		`</es:envelope>`

5	`json:object` (`map:map`) representation extracted from envelope document by `es:instance-from-document` or `es.instanceFromDocument`
	
	Shown here as JSON for readability, this is really a `map:map` in XQuery. In JavaScript, this function returns a JavaScript object. The value is mutable.
		`{ "id": "1234",`
		`  "firstName": "George",`
		`  "lastName": "Washington",`
		`  "fullName": "George Washington",`
		`  "$type": "Person"`
		`}`

	Representation	Example

6	XML representation extracted from envelope document by `es:instance-xml-from-document` or `es.instanceXmlFromDocument`
	
	The value is immutable.
		`<Person>`
		`  <id>1234</id>`
		`  <firstName>George</firstName>`
		`  <lastName>Washington</lastName>`
		`  <fullName>George Washington</fullName>`
		`</Person>`

7	JSON representation extracted from envelope document by `es:instance-json-from-document` or `es.instanceJsonFromDocument`
	
	This function returns a JSON object node. The value is immutable.
		`{ "Person": {`
		`  "id": "1234",`
		`  "firstName": "George",`
		`  "lastName": "Washington",`
		`  "fullName": "George Washington"`
		`} }`

The representations you see on lines 2, 3, and 4 were created by an instance converter module. For details, see “Creating an Instance Converter Module” on page 110. The representation on line 2 is a transient, mutable in-memory representation designed for ease of use in instance converter code. If you pass an envelope document to the `convert-instance-T` function of a version translator module, it returns a similar representation; for details, see “Creating a Model Version Translator Module” on page 116.

The envelope document representation on line 4 is the recommended way to store entity instances in MarkLogic. You can customize the contents of your envelope, but should usually leave the `es:instance` portion as-is. This is the layout produced by the `instance-to-envelope` function of an instance converter.

The representations on lines 5, 6, and 7 are instances extracted from an envelope document using the Entity Services API. The `map:map` representation on line 5 differs from the other extracted entities in that it is mutable and carries explicit type information in the `$type` property. This representation differs from the one on line 2 in that it contains only the instance entity type properties. There is no `$attachments`. For more details, see “Extracting an Entity Instance from an Envelope Document” on page 161.

5.1.3.2 JSON Entity Instance Representations

This example uses the `Person` entity type from the model defined in "Getting Started With Entity Services" on page 19.

	Representation	Example

1	Raw Source	`{ "pid": 2345,`
		`  "given": "Martha",`
		`  "family": "Washington"`
		`}`

2	In-memory instance, as returned by `extract-instance-Person`
	
	Shown here as JSON for readability, but really a `json:object` (`map:map`) with keys `$attachments`, `$type`, `id`, etc.
		`{ "$type": "Person",`
		`  "$attachments": {`
		`  "pid": 2345,`
		`  "given": "Martha",`
		`  "family": "Washington"`
		`  },`
		`  "id": 2345,`
		`  "firstName": "Martha",`
		`  "lastName": "Washington",`
		`  "fullName": "Martha Washington"`
		`}`

3	Canonical JSON instance generated by `instance-to-canonical`
	
	Used to construct the instance within an envelope document.
		`{"Person": {`
		`  "id":"2345",`
		`  "firstName": "Martha",`
		`  "lastName": "Washington",`
		`  "fullName": "Martha Washington"`
		`}}`

4	JSON Envelope document, as generated by `instance-to-envelope`	`{"envelope": {`
		`  "instance": {`
		`  "info": {`
		`  "title": "Person",`
		`  "version": "1.0.0"`
		`  },`
		`  "Person": {`
		`  "id": "2345",`
		`  "firstName": "Martha",`
		`  "lastName": "Washington",`
		`  "fullName": "Martha Washington"`
		`  }`
		`  },`
		`  "attachments": [ {`
		`  "pid": 2345,`
		`  "given": "Martha",`
		`  "family": "Washington"`
		`  } ]`
		`} }`

	Representation	Example

5	`json:object` (`map:map`) representation extracted from envelope document by `es:instance-from-document` or `es.instanceFromDocument`
	
	Shown here as JSON for readability, this is really a `map:map` in XQuery. In JavaScript, this function returns a JavaScript object. The value is mutable.
		`{ "$type": "Person",`
		`  "id":"2345",`
		`  "firstName":"Martha",`
		`  "lastName":"Washington",`
		`  "fullName":"Martha Washington"`
		`}`

6	XML representation extracted from envelope document by `es:instance-xml-from-document` or `es.instanceXmlFromDocument`
	
	The value is immutable.
		`<Person>`
		`  <id>2345</id>`
		`  <firstName>Martha</firstName>`
		`  <lastName>Washington</lastName>`
		`  <fullName>Martha Washington</fullName>`
		`</Person>`

7	JSON representation extracted from envelope document by `es:instance-json-from-document` or `es.instanceJsonFromDocument`
	
	This function returns a JSON object node. The value is immutable.
		`{ "Person": {`
		`  "id":"2345",`
		`  "firstName":"Martha",`
		`  "lastName":"Washington",`
		`  "fullName":"Martha Washington"`
		`}}`

The representations you see on lines 2, 3, and 4 were created by an instance converter module. For details, see “Creating an Instance Converter Module” on page 110. The representation on line 2 is a transient, mutable in-memory representation designed for ease of use in instance converter code. If you pass an envelope document to the `convert-instance-T` function of a version translator module, it returns a similar representation; for details, see “Creating a Model Version Translator Module” on page 116.

The envelope document representation on line 4 is the recommended way to store entity instances in MarkLogic. You can customize the contents of your envelope, but should usually leave the `instance` portion as-is. This is the layout produced by the `instance-to-envelope` function of an instance converter.

The representations on lines 5, 6, and 7 are instances extracted from an envelope document using the Entity Services API. The `map:map` representation on line 5 differs from the other extracted entities in that it is mutable and carries explicit type information in the `$type` property. This representation differs from the one on line 2 in that it contains only the instance entity type properties. There is no `$attachments` property. For more details, see "Extracting an Entity Instance from an Envelope Document" on page 161.

5.2 Creating an Entity Instance from a Data Source

The Entity Services API does not dictate how you create an entity instance from source data, but the recommended process is as follows:

- Generate, customize, and install an instance converter module, as described in "Creating an Instance Converter Module" on page 110.
- Use the `extract-instance-`*`T`* and `instance-to-envelope` functions of the instance converter module to create instance envelope documents for some entity type *`T`* from source data.
- Insert your envelope documents in the database.

By convention, instances are stored as child elements of an XML or JSON envelope document. You can extract an instance from an envelope document in several formats. For details, see "Extracting an Entity Instance from an Envelope Document" on page 161.

The following code illustrates one way to create envelope documents from raw source. In this example, the source data comes from documents in MarkLogic that are in a collection named "raw", and instances are generated for an entity type named `Person`. The generated envelope documents are in XML format; you could also choose JSON. This example uses the converter and data from "Getting Started With Entity Services" on page 19.

Language	Example

XQuery	`(: Create envelope documents from raw source documents :)`
	`xquery version "1.0-ml";`
	`import module namespace es =`
	`"http://marklogic.com/entity-services"`
	`  at "/MarkLogic/entity-services/entity-services.xqy";`
	`import module namespace person =`
	`  "http://example.org/example-person/Person-1.0.0"`
	`  at "/es-gs/person-1.0.0-conv.xqy";`
	
	`for $source in fn:collection('raw') return`
	`  let $instance := person:extract-instance-Person($source)`
	`  let $uri :=`
	`  fn:concat('/es-gs/env/', map:get($instance, 'id'), '.xml')`
	`  return xdmp:document-insert(`
	`  $uri,`
	`  person:instance-to-envelope($instance, "xml"),`
	`  <options xmlns="xdmp:document-insert">`
	`  <collections>`
	`  <collection>person-envelopes</collection>`
	`  </collections>`
	`  </options>`
	`  )`

JavaScript	`'use strict';`
	`declareUpdate();`
	`const es = require('/MarkLogic/entity-services/entity-services.xqy');`
	`const person = require('/es-gs/person-1.0.0-conv.xqy');`
	
	`for (const source of fn.collection('raw')) {`
	`  let instance = person.extractInstancePerson(source);`
	`  let uri = '/es-gs/env/' + instance.id + '.xml';`
	`  xdmp.documentInsert(`
	`  uri, person.instanceToEnvelope(instance, 'xml'),`
	`  {collections: ['person-envelopes']}`
	`  );`
	`}`

The resulting envelope documents have the following form by default. The instance data is accessible in an envelope document via the XPath expression `//es:instance` (or `//*:instance`). The original source from which the instance was derived is accessible via the XPath expression `//es:attachments` (or `//*:attachments`).

```
<es:envelope xmlns:es="http://marklogic.com/entity-services">
  <es:instance>
    <es:info>
      <es:title>Person</es:title>
      <es:version>1.0.0</es:version>
    </es:info>
    <Person>
      <id>1234</id>
      <firstName>George</firstName>
      <lastName>Washington</lastName>
      <fullName>George Washington</fullName>
    </Person>
  </es:instance>
  <es:attachments>
    <person>
      <pid>1234</pid>
      <given>George</given>
      <family>Washington</family>
    </person>
  </es:attachments>
</es:envelope>
```

If you generate JSON envelopes rather than XML envelopes, you get envelopes of the following form by default. The instance data is accessible in an envelope document via the XPath expression `//instance` (or `//*:instance`). The original source from which the instance was dervied is accessible via the XPath expression `//attachements` (or `//*:attachments`).

```
{ "envelope": {
  "instance": {
    "info": {
      "title": "Person",
      "version": "1.0.0"
    },
    "Person": {
      "id": "1234",
      "firstName": "George",
      "lastName": "Washington",
      "fullName": "George Washington"
    }
  },
  "attachments": [
    "<person><pid>1234<\/pid><given>George<\/given><family>Washington<\/famil
y><\/person>"
    ]
} }
```

Note: If your model specifies a namespace binding for an entity type and you use JSON envelopes, the namespace is discarded in the JSON representation, but the code and configuration artifacts still assumes a namespace, so it will not work properly with JSON envelope documents. You should use XML envelope documents for entity types that define a namespace binding.

For an end-to-end example of creating envelope documents using this model, see "Getting Started With Entity Services" on page 19.

5.3 Generating Test Entity Instances

You can generate test instances from a model using the `es:model-get-test-instances` XQuery function or `es.modelGetTestInstances` Server-Side JavaScript function. You can use test instances for tasks such as experimenting with model refinement and testing code that manipulates instances.

The test instances are based purely on the model and do not reflect data normalization or customization you add to your instance converter. The test instances can help you identify properties for which converter customization is required.

The `es:model-get-test-instances` and `es.modelGetTestInstances` functions return a sequence of instances, one for each entity type defined in the input model.

If an entity type property definition contains a local reference, the referenced entity type is assumed to be embedded in the referencing entity. If an entity type property definition contains an external reference, no meaningful test value can be generated.

For example, assume the following model defining two entity types, `Name` and `Person`. A `Person` contains a local reference to a `Name`.

```
{ "info": {
    "title": "Example",
    "version": "1.0.0",
    "description": "ES Examples"
  },
  "definitions": {
    "Name": {
      "properties": {
        "first": { "datatype": "string" },
        "last": { "datatype": "string" }
      }
    },
    "Person": {
      "properties": {
        "id": { "datatype": "int" },
        "name": { "$ref": "#/definitions/Name" },
      }
} } }
```

If you generate test instances from this model, the `name` property of the `Person` test instance contains a `Name` instance value:

```
<Person>
  <id>123</id>
  <name>
    <Name>
      <first>some string</first>
      <last>some string</last>
    </Name>
  </name>
</Person>
```

If the `name` property of a `Person` entity was an external reference to such as "http://example.com/SomeType" instead, then no meaningful test value can be generated. The `Person` test instance would look like the following:

```
<Person>
  <id>123</id>
  <name><SomeType>externally-referenced-instance</SomeType></name>
</Person>
```

To generate instances from real source data, use an instance converter. For more details, see "Creating an Instance Converter Module" on page 110 and "Creating an Entity Instance from a Data Source" on page 157.

5.4 Extracting an Entity Instance from an Envelope Document

Though Entity Services encourages storing your instances in MarkLogic in the form of envelope documents, downstream consumers of your data, such as client applications, will probably expect to receive the canonical instance data, not the entire envelope.

The Entity Services API includes the following XQuery functions for extracting an instance from an envelope document. The corresponding JavaScript functions follow.

XQuery Function	Extracted Instance Format
`es:instance-from-document`	`map:map` (`json:object`, mutable)
`es:instance-json-from-document`	`object-node()` (immutable)
`es:instance-xml-from-document`	`element()` (immutable)

The Entity Services API includes the following Server-Side JavaScript functions for extracting an instance from an envelope document.

JavaScript Function	Extracted Instance Format
`es.instanceFromDocument`	JavaScript object (mutable)
`es.instanceJsonFromDocument`	`object-node()` (immuntable)
`es.instanceXmlFromDocument`	`element()` (immuntable)

For example, suppose you have the following envelope document in the database with the URI `/es-gs/env/1234.xml`:

```
<es:envelope xmlns:es="http://marklogic.com/entity-services">
  <es:instance>
    <es:info>
      <es:title>Person</es:title>
      <es:version>1.0.0</es:version>
    </es:info>
    <Person>
      <id>1234</id>
      <firstName>George</firstName>
      <lastName>Washington</lastName>
      <fullName>George Washington</fullName>
    </Person>
  </es:instance>
  <es:attachments>
    <person>
      <pid>1234</pid>
      <given>George</given>
      <family>Washington</family>
    </person>
  </es:attachments>
</es:envelope>
```

Then, the following code snippet extracts an instance from the envelope document as a `json:object` in XQuery or a JavaScript object in JavaScript.

Language	Example

XQuery	`xquery version "1.0-ml";`
	`import module namespace es =`
	`"http://marklogic.com/entity-services"`
	`  at "/MarkLogic/entity-services/entity-services.xqy";`
	
	`es:instance-from-document(fn:doc('/es-gs/env/1234.xml'))[1]`

JavaScript	`'use strict';`
	`const es =`
	`require('/MarkLogic/entity-services/entity-services.xqy');`
	
	`fn.head(`
	`  es.instanceFromDocument(cts.doc('/es-gs/env/1234.xml'))`
	`);`

The result is a sequence containing one item, equivalent to the following JSON:

```
{ "id":"1234",
  "firstName":"George",
  "lastName":"Washington",
  "fullName":"George Washington",
  "$type": "Person"
}
```

The following table illustrates the result of calling each of the instance envelope extraction functions.

Function	Result

`es:instance-from-document`
 	
`es.instanceFromDocument`
	A `json:object` (XQuery) or JavaScript object (JavaScript) equivalent to the following:
	`{ "id":"1234",`
	`"firstName":"George",`
	`"lastName":"Washington",`
	`"fullName":"George Washington",`
	`"$type":"Person"`
	`}`

`es:instance-json-from-document`
 	
`es.instanceJsonFromDocument`
	A JSON `object-node()` equivalent to the following:
	`{ "Person": {`
	`"id":"1234",`
	`"firstName":"George",`
	`"lastName":"Washington",`
	`"fullName":"George Washington"`
	`}`

`es:instance-xml-from-document`
 	
`es.instanceXmlFromDocument`
	The following XML element:
	`<Person xmlns:es=...>`
	`<id>1234</id>`
	`<firstName>George</firstName>`
	`<lastName>Washington</lastName>`
	`<fullName>George Washington</fullName>`
	`</Person>`

For more detailed coverage of instance representations, see “What is an Instance?” on page 149 and “Example: Entity Instance Representations” on page 152.

5.5 Extracting the Original Source from an Envelope Document

If you follow the Entity Services conventions, an envelope document encapsulates both the canonical instance data and the raw source from which it was derived. This encapsulation happens when you call the `instance-to-envelope` XQuery function in a model’s generated instance converter module.

You can extract the attachments from an envelope document using the `es:instance-get-attachments` XQuery function or the `es.instanceGetAttachments` JavaScript function. You can use these function on a customized envelope, as long as the attacments are locatable via the XPath expression `//es:attachments`.

The raw source data is saved in the envelope as an attachment. For example, the highlighted `<person/>` element below is the raw XML source from which the enveloped instance was derived.

```
<es:envelope xmlns:es="http://marklogic.com/entity-services">
  <es:instance>...</es:instance>
  <es:attachments>
    <person>
      <pid>1234</pid>
      <given>George</given>
      <family>Washington</family>
    </person>
  </es:attachments>
</es:envelope>
```

If the format of the source data does not match the format of the envelope, the source data is serialized and stored in the envelope as a string. For example, if the source data is JSON and the envelope value is XML, then the source is stored as the text value of an `es:attachments` XML element. The following snippet is from an XML envelope document created from JSON source:

```
<es:envelope xmlns:es="http://marklogic.com/entity-services">
  <es:instance>...</es:instance>
  <es:attachments>{"pid":2345, "given":"Martha",
"family":"Washington"}</es:attachments>
</es:envelope>
```

The following code extracts the raw source attachment from an envelope document, assuming it is the only attachment.

Language	Example

XQuery	`xquery version "1.0-ml";`
	`import module namespace es =`
	`"http://marklogic.com/entity-services"`
	`  at "/MarkLogic/entity-services/entity-services.xqy";`
	
	`es:instance-get-attachments(fn:doc('/es-gs/env/1234.xml'))[1]`

JavaScript	`'use strict';`
	`const es =`
	`require('/MarkLogic/entity-services/entity-services.xqy');`
	
	`fn.head(`
	`  es.instanceGetAttachments(cts.doc('/es-gs/env/2345.xml'))`
	`);`

If there are multiple children in the `//es:attachments element`, you are responsible for picking out the raw source from the other attachments. There will only be multiple attachments if you explicitly add extra attachments.

If the original source attachment and the envelope format do not match, you must convert the serialization if you want to work with the data in its original form. For example, the following code deserializes a serialized JSON attachment from an XML envelope document, and then accesses one of its properties.

Language	JSON Deserialization Example

XQuery	`xquery version "1.0-ml";`
	`import module namespace es =`
	`"http://marklogic.com/entity-services"`
	`  at "/MarkLogic/entity-services/entity-services.xqy";`
	
	`map:get(`
	`  xdmp:from-json-string(`
	
	`es:instance-get-attachments(fn:doc('/es-gs/env/2345.xml'))[1]`
	`  )[1], "pid"`
	`)`

Server-Side JavaScript	`'use strict';`
	`const es =`
	`require('/MarkLogic/entity-services/entity-services.xqy');`
	
	`fn.head(xdmp.fromJsonString(`
	`  fn.head(`
	`  es.instanceGetAttachments(cts.doc('/es-gs/env/2345.xml')))`
	`)).pid;`

The following code is a similar example that extracts an XML attachment from a JSON envelope:

Language	XML Deserialization Example

XQuery	`xquery version "1.0-ml";`
	`import module namespace es =`
	`"http://marklogic.com/entity-services"`
	`  at "/MarkLogic/entity-services/entity-services.xqy";`
	
	`xdmp:unquote(`
	`  es:instance-get-attachments(fn:doc('/es-gs/env/1234.json'))[1]`
	`)[1]//pid/data()`

Server-Side JavaScript	`'use strict';`
	`const es =`
	`require('/MarkLogic/entity-services/entity-services.xqy');`
	
	`fn.head( xdmp.unquote(`
	`  fn.head(es.instanceGetAttachments(cts.doc('/es-gs/env/1234.json')))`
	`)).xpath('//pid/data()')`

5.6 Updating Entity Instance Data When Your Model Changes

As your model changes, you might need to update your instance data to match. Model changes can also impact generated and configuration artifacts. For details, see “Managing Model Changes” on page 88.

6.0 Querying a Model or Entity Instances

This chapter contains the following topics related to searching entity instances and models using MarkLogic. Unless otherwise noted, all the examples in this chapter use the entity model and data from “Getting Started With Entity Services” on page 19.

This chapter covers the following topics:

Additional examples are available in the Entity Services GitHub repository. For more details, see “Exploring the Entity Services Open-Source Examples” on page 15.

6.1 Query Support Provided by Entity Services

The Entity Services API includes the following utility functions that make it easier to create and configure an application that searches entity models and entity instances.

- Use `es:database-properties-generate` (XQuery) or `es.databasePropertiesGenerate` (JavaScript) to create a database configuration artifact with which to configure database range indexes and lexicons. This function relies on the model descriptor to identify properties that should be indexed or cataloged in a lexicon. For details, see “Generating a Database Configuration Artifact” on page 137.
- Use `es:search-options-generate` (XQuery) or `es.searchOptionsGenerate` (JavaScript) to generate a set of query options suitable for use with the Search API and the Client APIs. Some of the generated options rely on the model descriptor to identify properties that should be indexed or cataloged in a lexicon. For details, see “Generating Query Options for Searching Instances” on page 141.

- Use `es:extraction-template-generate` (XQuery) or `es.extractionTemplateGenerate` (JavaScript) to create a TDE template to enable querying instances as semantic or row data. For details, see "Generating a TDE Template" on page 122.

You can customize all of these generated artifacts to suit the requirements of your application.

You are not required to generate and use any of these artifacts, but doing so can make it easier to build a search application around your model. The examples in this chapter take advantage of these artifacts where appropriate.

6.2 Search Basics for Models

You can use Semantic search to search and make inferences about a model.

Recall that when you persist a model descriptor as part of the special Entity Services collection, MarkLogic generates a set of facts that define the core of your model, expressed as semantic triples. You can also enrich your model with additional facts (triples) that are not derivable from the model descriptor. For details, see "Introduction" on page 57.

The auto-generated triples include facts such as the following. For the complete ontology, see *MARKLOGIC_INSTALL_DIR*`/Modules/MarkLogic/entity-services/entity-services.ttl`.

- Model M defines entity type T
- Entity type T has a property P
- Property P of entity type T has data type D
- Entity Type T has primary key P

You can inspect all the triples associated with a model by evaluating a SPARQL query such as the following in Query Console:

XQuery	Server-Side JavaScript

`xquery version "1.0-ml";`
`cts:triples(`
`  (), (), (), (), (),`
`  cts:document-query(`*`yourModelURI`*`))`
	`'use strict';`
	`cts.triples(`
	`  null, null, null, null, null,`
	`  cts.documentQuery(`*`yourModelURI`*`));`

You can use SPARQL or the Optic API to perform a semantic search of the model. The following interfaces accept SPARQL input:

- The `sem:sparql` XQuery function or the `sem.sparql` Server-Side JavaScript function.
- The REST, Java, and Node.js client APIs accept SPARQL queries as input to their search interfaces. You can embed a SPARQL query in a combined query, or use an appropriate Java or Node.js query builder.

- Evaluate SPARQL directly in Query Console during development.

For a server-side model query example, see “Example: Using SPARQL for Model Queries” on page 174. For the Client APIs, refer to the respective developer guides listed in “Where to Find Additional Information” on page 193.

The Optic API enables semantic queries directly using JavaScript and XQuery, without requiring you to use a secondary query language (SPARQL). You can use the Optic API to query your model server-side using the `op:from-triples` XQuery function or the `op.fromTriples` Server-Side JavaScript function. For more details, see “Optic API for Multi-Model Data Access” on page 305 in the *Application Developer's Guide*.

6.3 Search Basics for Instance Data

You can query your instance data as documents, rows, or triples. See the following topics for more details:

- Document Search
- Row Search
- Semantic Search

Document search is always available. Row and semantic search are only available if you generate and install a TDE template, as described in “Generating a TDE Template” on page 123. In addition, semantic search is only available if an entity type defines a primary key.

6.3.1 Document Search

If you follow the Entity Services conventions, your instance data, as well as original source data is stored in envelope documents. The default structure of envelope documents is covered in “What is an Envelope Document?” on page 150.

You can use any of the available document search interfaces to search your envelope documents. For example:

- The `cts:search` XQuery function or `cts.search` Server-Side JavaScript Function. See “Example: Using cts:query or cts.query for Instance Queries” on page 175.
- The Server-Side JavaScript JSearch API. See “Example: Using JSearch for Instance Queries” on page 179.
- The XQuery Search API (`search:search`). See “Example: Using the Search API for Instance Queries” on page 176.
- The REST, Java, and Node.js Client APIs. See “Example: Using the Client APIs for Instance Queries” on page 180.

To learn more about any of these interfaces, see the links in “Where to Find Additional Information” on page 193.

The Search API and the Client APIs can take advantage of the query options you can generate using the Entity Services API. These options can help streamline and customize your searches. See the examples and "Generating Query Options for Searching Instances" on page 141.

You can also generate a database configuration artifact based on your model. The artifact includes index configuration for selected properties identified in the model. Creating these indexes can enhance search performance. For details, see "Generating a Database Configuration Artifact" on page 137.

6.3.2 Row Search

You can search your entity instance data as rows if you generate and install a TDE template based on your model. Broadly speaking there is an implicit table that corresponds to each entity type, with a row for each instance and columns for each property. For more details, see "Generating a TDE Template" on page 122.

You can use SQL or the Optic API to search your entities as rows using the following interfaces:

- The `xdmp:sql` XQuery function and the `xdmp.sql` Server-Side JavaScript function accept SQL input directly. See "Example: Using SQL for Instance Queries" on page 190.
- The Optic API `op:from-view` XQuery function and `op.fromView` Server-Side JavaScript function enable you to build and execute a query plan based on a row-oriented view of your data. See "Example: Using the Optic API for Instance Queries" on page 191.
- The Java Client API. Use the `com.marklogic.client.row.RowManager` interface and `com.marklogic.client.expression.PlanBuilder` class to build and evaluate an Optic row-based or triples-based query plan. For details, see Optic Java API for Relational Operations in the *Java Application Developer's Guide*.
- The REST Client API `/rows` service enables you to execute an Optic row-based or triples-based query plan. For details, see `GET:/v1/rows` or `POST:/v1/rows` in the *MarkLogic REST API Reference*.

You can also evaluate SQL directly in Query Console during development.

For more information about these interfaces, see the resources listed in "Where to Find Additional Information" on page 193.

6.3.3 Semantic Search

You can search your entity instances using semantic queries if and only if all of the following conditions are met:

- The entity type definition defines a primary key. A primary key enables unique identification of each instance. For details, see "Identifying the Primary Key Entity Property" on page 67.

- You generate and install a TDE template as described in “Generating a TDE Template” on page 122.

When these requirements are met, MarkLogic automatically generates a few facts about each instance when you insert an envelope document into the database. The facts take the form of semantic triples, which you can query using SPARQL or the Optic API. You can also extend the TDE template to include your own triples.

For an example of semantic queries on instance data, see “Example: Using SPARQL for Instance Queries” on page 189 and “Example: Using the Optic API for Instance Queries” on page 191.

You can use the following interfaces to perform a semantic search of your entity instance data:

- The `sem:sparql` XQuery function or the `sem.sparql` Server-Side JavaScript function. See “Example: Using SPARQL for Instance Queries” on page 189.
- The `op:from-triples` XQuery function or the `op.fromTriples` Server-Side JavaScript function of the Optic API. See “Example: Using the Optic API for Instance Queries” on page 191.
- Pass a SPARQL query to MarkLogic using the REST, Java, or Node.js client APIs. You can embed a SPARQL query in a combined query, or use an appropriate Java or Node.js query builder.
- The Java Client API. Use the `com.marklogic.client.row.RowManager` interface and `com.marklogic.client.expression.PlanBuilder` class to build and evaluate an Optic row-based or triple-based query plan. For details, see Optic Java API for Relational Operations in the *Java Application Developer’s Guide*.
- The REST Client API `/rows` service enables you to execute an Optic row-based or triples-based query plan. For details, see `GET:/v1/rows` or `POST:/v1/rows` in the *MarkLogic REST API Reference*.

You can also evaluate SPARQL directly in Query Console during development.

To learn more about these interfaces, see the resources listed in “Where to Find Additional Information” on page 193.

6.4 Pre-Installing Query Options

Recall that you can generate and customize model-specific query options for use with the Search API and the REST, Java, and Node.js Client APIs; see “Generating Query Options for Searching Instances” on page 141.

You must pre-install these options on MarkLogic if and only if all the following are true:

- You search your model or entity instances using one of the Client APIs (REST, Java, or Node.js).
- You do not want to specify options dynamically at query time, such as in a combined query.

You can install query options using the REST and Java Client APIs. For details, see the following topics:

- REST Client API: Creating or Modifying Query Options in the *REST Application Developer's Guide*
- Java Client API: Creating Persistent Query Options From Raw JSON or XML in the *Java Application Developer's Guide*

You can use persistent query options with the Node.js Client API, but you cannot install them. Use REST or Java instead.

6.5 Example: Using SPARQL for Model Queries

When you insert a model descriptor document into MarkLogic as part of the special Entity Services collection, MarkLogic creates a model from the descriptor. The model is expressed as semantic triples; for details, see "Search Basics for Models" on page 170.

You can also extend the model with your own triples; for details, see "Extending a Model with Additional Facts" on page 87.

You can query triples in MarkLogic using the following APIs:

- The `sem:sparql` XQuery function or the `sem.sparql` Server-Side JavaScript functions.
- The Client APIs; see Client-Side APIs for Semantics in the *Semantics Developer's Guide*.
- The Optic API XQuery; see the `op:from-triples` XQuery function or the `op.fromTriples` JavaScript function.

The following SPARQL query returns the name of all required properties of the `Person` entity type of the model created in "Getting Started With Entity Services" on page 19.

```
prefix es:<http://marklogic.com/entity-services#>
select ?ptitle
where {
  ?x a es:EntityType;
       es:title "Person";
       es:property ?property .
  ?property a es:RequiredProperty;
              es:title ?ptitle
}
```

If you run this query in Query Console against the data from "Getting Started With Entity Services" on page 19, it will return the property names "lastName", "firstName", and "fullName".

The following example uses `sem:sparql` or `sem.sparql` to evaluate the same SPARQL query.

Language	Example

XQuery	`xquery version "1.0-ml";`
	`sem:sparql('`
	`  prefix es:<http://marklogic.com/entity-services#>`
	`  select ?ptitle`
	`  where {`
	`  ?x a es:EntityType;`
	`  es:title "Person";`
	`  es:property ?property .`
	`  ?property a es:RequiredProperty;`
	`  es:title ?ptitle`
	`  }'`
	`)`

JavaScript	`sem.sparql(`
	`  'prefix es:<http://marklogic.com/entity-services#> ' +`
	`  'select \?ptitle ' +`
	`  'where {' +`
	`  '?x a es:EntityType;' +`
	`  'es:title "Person";' +`
	`  'es:property ?property .' +`
	`  '?property a es:RequiredProperty;' +`
	`  'es:title ?ptitle' +`
	`  '}'`
	`)`

6.6 Example: Using cts:query or cts.query for Instance Queries

The cts query interface serves as the foundation for most higher level document search APIs in MarkLogic. Using the cts layer gives you fine-grained control over your searches while the XQuery Search API, JavaScript JSearch API, and the Client APIs provide higher level abstractions on top of this layer. For details, see APIs for Multiple Programming Languages in the *Search Developer's Guide*.

The following example uses the `cts:search` XQuery function or `cts.search` JavaScript function to find all `Person` envelope documents where the instance data includes a "lastName" element with the value "washington". For the sake of simplicity, the example prints out just the value of the "fullName" property in the matched documents, rather than complete documents.

Language	Example

XQuery	`xquery version "1.0-ml";`
	
	`cts:search(fn:collection('person-envelopes'),`
	`  cts:element-query(`
	`  fn:QName("http://marklogic.com/entity-services", "instance"),`
	`  cts:element-value-query(xs:QName("lastName"), "washington")`
	`  )`
	`)//fullName/fn:data()`

JavaScript	`const results = cts.search(cts.andQuery((`
	`  cts.collectionQuery('person-envelopes'),`
	`  cts.elementQuery(`
	`  fn.QName('http://marklogic.com/entity-services', 'instance'),`
	`  cts.elementValueQuery(xs.QName('lastName'), 'washington')`
	`  )`
	`)));`
	
	`// Accumulate the matched names in an array for easy display`
	`// in Query Console.`
	`const names = [];`
	`for (const doc of results) {`
	`  names.push(doc.xpath('//Person/fullName/fn:data()'));`
	`}`
	`names`

You could also use a path query instead of an element query to limit the search to `es:instance` elements.

If you run the example code in Query Console against the envelope documents created in "Getting Started With Entity Services" on page 19, the results are "George Washington" and "Martha Washington".

6.7 Example: Using the Search API for Instance Queries

The XQuery Search API is an interface that abstracts away some of the complexity of `cts:search` operations such as the generation of facets and snippets. For details, see Search API: Understanding and Using in the *Search Developer's Guide*.

Server-Side JavaScript developers should use the JSearch API instead of the XQuery Search API. You can use the Search API from JavaScript, but the search configuration and results are expressed in XML, so it is not as convenient or "natural". See "Example: Using JSearch for Instance Queries" on page 179, instead.

Recall that you can generate Search API compatible query options using the Entity Services API; for details, see “Generating Query Options for Searching Instances” on page 141. The code samples in this section assume you generated options from the model in “Getting Started With Entity Services” on page 19. To learn more about the generated options, see “Characteristics of the Generated Options” on page 142.

The following example uses generated options to find all `Person` envelope documents where the instance data includes the word “washington”. For simplicity, only the value of the “fullName” property is displayed. (In practice, you would probably customize the generated options for your application.)

```
xquery version "1.0-ml";
import module namespace search =
  "http://marklogic.com/appservices/search"
  at "/MarkLogic/appservices/search/search.xqy";
import module namespace es = "http://marklogic.com/entity-services"
  at "/MarkLogic/entity-services/entity-services.xqy";

let $options := es:search-options-generate(
  fn:doc('/es-gs/models/person-1.0.0.json'))
let $matches :=
  search:search("entity-type:Person AND washington", $options)
return $matches//Person/fullName/fn:data()
```

If you run this code in Query Console against the envelope documents created in “Getting Started With Entity Services” on page 19, then you should see output similar to the following:

```
Martha Washington
George Washington
```

The search term “`entity-type:Person`” constrains the search to `Person` entities. The `entity-type` constraint is automatically generated for all models.

The generated options also include an `additional-query` option that constrains results to the instance data in an envelope document. For example:

```
<search:constraint name="entity-type">
  <search:value>
    <search:element ns="http://marklogic.com/entity-services" name="title"/>
  </search:value>
</search:constraint>

<search:additional-query>
  <cts:element-query xmlns:cts="http://marklogic.com/cts">
    <cts:element xmlns:es="...">es:instance</cts:element>
    <cts:true-query/>
  </cts:element-query>
</search:additional-query>
```

Though the code above returns just the value of the "fullName" property in each matched instance, the search results contain the entire entity, as if you called `es:entity-from-document` on the envelope document. This data is contained in the `search:extracted` element of each `search:result`. For example:

```
<search:response snippet-format="empty-snippet" total="2" start="1"
      page-length="10" selected="include" xmlns:search=...>
  <search:result index="1" uri="/es-gs/env/2345.xml"
      path="fn:doc(&amp;quot;/es-gs/env/2345.xml&amp;quot;)" score="15872"
      confidence="0.4703847" fitness="0.7823406">
    <search:snippet/>
    <search:extracted kind="element">
      <Person>
        <id>2345</id>
        <firstName>Martha</firstName>
        <lastName>Washington</lastName>
        <fullName>Martha Washington</fullName>
      </Person>
    </search:extracted>
  </search:result>
  <search:result .../>
  <search:qtext>entity-type:Person AND washington</search:qtext>
  <search:metrics>...</search:metrics>
</search:response>
```

The generated options enable this behavior by disabling snippeting and faceting, and defining an `extract-document-data` option that extracts just the instance from the envelope document. For example:

```
<search:extract-document-data selected="include">
  <search:extract-path
xmlns:es=...>//es:instance/(Person)</search:extract-path>
</search:extract-document-data>

<search:additional-query>
  <cts:element-query xmlns:cts="http://marklogic.com/cts">
    <cts:element xmlns:es=...>es:instance</cts:element>
    <cts:true-query/>
  </cts:element-query>
</search:additional-query>

<search:return-facets>false</search:return-facets>
<search:transform-results apply="empty-snippet"/>
```

If the model included more than one entity type definition, then the `extract-document-data` option would use an extract path that matched any of the defined types. For example, if the model defines a second entity type named "Family", then the extract path would be the following:

```
//es:instance/(Family|Person)
```

If an entity type definition includes range index or word lexicon specifications, then the options would include additional range or word constraints options. For example, if we extend the `Person` entity to include a "rating" property of type float with a `pathRange-index` specification, then the generated options would include a path range constraint similar to the following:

```
<search:constraint name="rating">
  <search:range type="xs:float" facet="true">
    <search:path-index xmlns:es="http://marklogic.com/entity-services">
        //es:instance/Person/rating
    </search:path-index>
  </search:range>
</search:constraint>
```

This enables a query string such as "entity-type:Person AND rating GT 3.0".

For an example of a complete set of generated options, see "Example: Generating Query Options" on page 144.

To learn more about query options, see Search Customization Using Query Options and Appendix: Query Options Reference in the *Search Developer's Guide*.

6.8 Example: Using JSearch for Instance Queries

The JSearch API is a fluent Server-Side JavaScript search interface. You can use it to search documents using a variety of query styles, as well as for querying lexicons and range indexes. For details, see Creating JavaScript Search Applications in the *Search Developer's Guide*.

The following example use a `cts.query` to find all `Person` envelope documents where the instance data includes a "lastName" element with the value "washington". For the sake of display simplicity, a custom mapper is used to extract just the value of the "fullName" property from each matched instance, instead of returning full search results.

```
'use strict';
const jsearch = require('/MarkLogic/jsearch.sjs');

jsearch.collections('person-envelopes').documents()
  .where(cts.elementQuery(
      fn.QName('http://marklogic.com/entity-services', 'instance'),
      cts.elementValueQuery('lastName', 'washington')))
  .map(function(match) {
         return match.document.xpath('//fullName/fn:data()');
       })
  .result();
```

If you run the example in Query Console against the envelope documents created in "Getting Started With Entity Services" on page 19, the results should be similar to the following:

```
{ "results":[
    "Martha Washington",
```

```
    "George Washington"],
  "estimate":2}
```

6.9 Example: Using the Client APIs for Instance Queries

This section provides examples of querying instances with the REST, Java, and Node.js Client APIs. Note that these APIs support more query styles than are shown here. For details, refer to the development guide for each API. These guides are listed in “Where to Find Additional Information” on page 193.

- Java Client API
- Node.js Client API
- REST Client API

6.9.1 Java Client API

The Java Client API is an API for creating client applications that interact with MarkLogic. The API enables you to search documents using a variety of query styles. For more details, see the *Java Application Developer's Guide* and the *Java Client API Documentation*. The Java Client API can take advantage of the Search API compatible query options you can generate with the Entity Services API, as discussed in “Generating Query Options for Searching Instances” on page 141.

The following example uses a string query to find all `Person` envelope documents where the instance data includes the word “washington”. The code assumes you have already generated query options using the Entity Services API and installed them on MarkLogic as persistent query options under the name `OPTIONS_NAME`; see the complete example below for an example of how to install the options.

```
QueryManager qm = client.newQueryManager();
StringQueryDefinition query =
    qm.newStringDefinition(OPTIONS_NAME)
      .withCriteria("entity-type:Person AND washington");
SearchHandle results = qm.search(query, new SearchHandle());
```

For a discussion of how the generated options enable this query string, see “Example: Using the Search API for Instance Queries” on page 176.

You could also create a `RawCombinedQueryDefinition` and embed the generated options inside the combined query. This enables you to use the generated options without first persisting them on MarkLogic. For more details, see Apply Dynamic Query Options to Document Searches in the *Java Application Developer's Guide*.

The following code is a complete example of installing options and performing the above search. This code installs the query options (if necessary), performs the search, and prints out the value of the fullName property in the matched entities.

Note: Modify the values in bold to fit your environment.

```
package examples;

import java.io.File;

import com.marklogic.client.DatabaseClient;
import com.marklogic.client.DatabaseClientFactory;
import com.marklogic.client.admin.QueryOptionsManager;
import com.marklogic.client.io.FileHandle;
import com.marklogic.client.io.Format;
import com.marklogic.client.io.QueryOptionsListHandle;
import com.marklogic.client.io.SearchHandle;
import com.marklogic.client.query.ExtractedItem;
import com.marklogic.client.query.ExtractedResult;
import com.marklogic.client.query.MatchDocumentSummary;
import com.marklogic.client.query.QueryManager;
import com.marklogic.client.query.StringQueryDefinition;

import javax.xml.xpath.XPathExpression;
import javax.xml.xpath.XPathExpressionException;
import javax.xml.xpath.XPathFactory;

import org.w3c.dom.Document;


public class EntityServices {
    private static DatabaseClient client =
        DatabaseClientFactory.newClient(
            "localhost", 8000, "es-gs",
            new DatabaseClientFactory.DigestAuthContext(USER, PASSWORD));
    static String OPTIONS_NAME = "person-1.0.0";
    static String OPTIONS_PATHNAME =
        "/path/to/options/person-options-1.0.0.xml";

    // Install the options generated by ES, if needed.
    public static void installOptions(String filename, String optionsName) {
        QueryOptionsManager optMgr =
            client.newServerConfigManager()
                  .newQueryOptionsManager();
        QueryOptionsListHandle optList =
            optMgr.optionsList(new QueryOptionsListHandle());

        if (optList.getValuesMap().get(OPTIONS_NAME) == null) {
            FileHandle options =
                new FileHandle(new File(filename))
                    .withFormat(Format.XML);
            optMgr.writeOptions(optionsName, options);
        }
    }

    public static void main(String[] args) throws XPathExpressionException {
        // Install the options generated by ES, if necessary
```

```
            installOptions(OPTIONS_PATHNAME, OPTIONS_NAME);

            // Build the query
            QueryManager qm = client.newQueryManager();
            StringQueryDefinition query =
                qm.newStringDefinition(OPTIONS_NAME)
                  .withCriteria("entity-type:Person AND washington");

            // Perform the search
            SearchHandle results = qm.search(query, new SearchHandle());

            // Iterate over the results, and write out just the value of
            // the "fullName" property.
            XPathExpression xpath =
                XPathFactory.newInstance().newXPath().compile("//fullName");
            for (MatchDocumentSummary match : results.getMatchResults()) {
                ExtractedResult extracted = match.getExtracted();
                for (ExtractedItem item : extracted) {
                    Document person = item.getAs(Document.class);
                    System.out.println(xpath.evaluate(person));
                }
            }

            client.release();
        }
    }
```

If you run this example, it will print the values “Martha Washington” and “George Washington”.

As discussed in “Example: Using the Search API for Instance Queries” on page 176, the matched entities are returned as extracted items in the search response. The following part of the example iterates over the search results, accesses the extracted entity data, and then prints out just the value of the `fullName` property. The `person` variable holds the entity, as a DOM Document.

```
XPathExpression xpath =
    XPathFactory.newInstance().newXPath().compile("//fullName");
for (MatchDocumentSummary match : results.getMatchResults()) {
    ExtractedResult extracted = match.getExtracted();
    for (ExtractedItem item : extracted) {
        Document person = item.getAs(Document.class);
        System.out.println(xpath.evaluate(person));
    }
}
```

6.9.2 Node.js Client API

The Node.js Client API enables you to create Node.js client applications that interact with MarkLogic. The API enables you to search documents using a variety of query styles. For more details, see the *Node.js Application Developer’s Guide* and the *Node.js API Reference*.

Recall that you can generate Search API compatible query options using the Entity Services API; for details, see “Generating Query Options for Searching Instances” on page 141. You can only take advantage of these options if you pre-install them as described in “Pre-Installing Query Options” on page 173 and then reference them in a combined query.

However, you can use the Node.js query builder to create equivalent behavior without using the generated options. This section explores both approaches:

- Search Using Pre-Installed Options
- Search Without Pre-Installing Options

6.9.2.1 Search Using Pre-Installed Options

This example uses a combined query and pre-installed query options. The example assumes you generated options from the model in “Getting Started With Entity Services” on page 19, and then installed the options on MarkLogic with the name “person-1.0.0”. You can install the options using the REST Client API or Java Client API; for details, see “Pre-Installing Query Options” on page 173.

The following example finds all `Person` envelope documents where the instance data includes the word “washington”. The search returns just the matched instance data, as serialized XML.

```
const marklogic = require('marklogic');

// MODIFY THIS VAR TO MATCH YOUR ENV
const connInfo = {
    host: 'localhost',
    port: 8000,
    user: 'username',
    password: 'password',
    database: 'es-gs'
  };
const db = marklogic.createDatabaseClient(connInfo);
const qb = marklogic.queryBuilder;

// entity-type is a constraint defined by the options.
// The options should already be installed, with name 'person-1.0.0'.
const combinedQuery = {
  search: {
    query: 'entity-type:Person AND washington'
  },
  optionsName: 'person-1.0.0'
};

db.documents.query(
  { search: {
      qtext: 'entity-type:Person AND washington'
    },
    optionsName: 'person-1.0.0'
  }
```

```
).result( function(results) {
  for (let result of results) {
    console.log(JSON.stringify(result.content));
  }
});
```

The query matches entities with “fullName” property values of “Martha Washington” and “George Washington”. The options limit the returned data to just the matched entities through the `extract-document-data` option. Since the envelope documents are XML, each extracted entity is returned as a string containing serialized XML, with a root element of `<search:extracted/>`. For example, the result for “Martha Washington” looks like the following. (Line breaks have been added for readability; the value of the “content” property is one string.)

```
{ "uri":"/es-gs/env/2345.xml",
  "category":"content",
  "format":"xml",
  "contentType":"application/xml",
  "contentLength":"394",
  "content":
    "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n
     <search:extracted kind=\"element\" format=\"xml\"
       context=\"fn:doc(&quot;/es-gs/env/2345.xml&quot;)\"
       xmlns:search=\"http://marklogic.com/appservices/search\">
      <Person>
        <id>2345</id>
        <firstName>Martha</firstName>
        <lastName>Washington</lastName>
        <fullName>Martha Washington</fullName>
      </Person>
    </search:extracted>"}
```

6.9.2.2 Search Without Pre-Installing Options

The following example uses the Node.js `queryBuilder` interface to perform a search equivalent to “Search Using Pre-Installed Options” on page 183. This approach requires a more in-depth understanding of the relationship between the builder interface and the underlying Search API query options.

Before you can run this example, you must configure the REST Client API instance through which you connect to MarkLogic so that it defines a namespace binding for the prefix “es”. The binding is required because the example uses `queryBuilder.extract` to extract just the `es:instance` portion of an envelope document.

The Node.js Client API does not directly support namespace binding configuration, so this example uses the REST Client API and the `curl` command line tool to do so. For more details, see Using Namespace Bindings in the *REST Application Developer’s Guide*. You can replace the use of curl with any tool that can send HTTP requests.

Run the following command to define a binding between the “es” prefix and the “http://marklogic.com/entity-services”. Change the user, password, host, and port as needed to match your environment.

```
# Windows users, see Modifying the Example Commands for Windows
curl --anyauth --user user:password -X PUT \
  -d '{ "prefix": "es", "uri": "http://marklogic.com/entity-services" }' \
  -H "Content-type: application/json" -i \
  http://localhost:8000/v1/config/namespaces/es
```

If the command is successful, MarkLogic returns a `201 Created` status.

The following Node.js script finds all `Person` envelope documents where the instance data includes the word “washington”. The search returns just the matched instance data, as serialized XML. A discussion of the relationship between the built query below and the generated query options follows.

```
const marklogic = require('marklogic');

// MODIFY THIS VAR TO MATCH YOUR ENV
const connInfo = {
    host: 'localhost',
    port: 8000,
    user: 'username',
    password: 'password',
    database: 'es-gs'
  };
const db = marklogic.createDatabaseClient(connInfo);
const qb = marklogic.queryBuilder;

db.documents.query(
  qb.where(
    qb.collection('person-envelopes'),
    qb.scope(
      qb.element(
        qb.qname('http://marklogic.com/entity-services','instance')),
      qb.and()),
    qb.parsedFrom('entity-type:person AND washington',
      qb.parseBindings(
        qb.value(
          qb.element(
            qb.qname('http://marklogic.com/entity-services','title')),
          qb.bind('entity-type'))))
  ).slice(qb.extract({
       paths: ['//es:instance/(Person)'],
       selected: 'include'
     }))
).result( function(results) {
  for (let result of results) {
    console.log(JSON.stringify(result.content));
  }
});
```

Use `qb.scope` to create a container query that mimics the generated `additional-query` option restricting results to matches within `es:instance` elements.

Description	Example

Generated Option	`<search:additional-query>`
	`  <cts:element-query xmlns:cts="...">`
	`  <cts:element xmlns:es="http://marklogic.com/entity-services">`
	`  es:instance`
	`  </cts:element>`
	`  <cts:true-query/>`
	`  </cts:element-query>`
	`</search:additional-query>`

Node.js Equivalent	`qb.scope(`
	`  qb.element(`
	`  qb.qname('http://marklogic.com/entity-services',`
	`  'instance')),`
	`  qb.and())`

Use a parse binding to bind the tag “entity-type” to the title element of an entity instance so that you can constrain string queries to specific entity types. The bind enables search terms such as “entity-type:Person”.

Description	Example

Generated Option	`<search:constraint name="entity-type">`
	`  <search:value>`
	`  <search:element ns="http://marklogic.com/entity-services"`
	`  name="title"/>`
	`  </search:value>`
	`</search:constraint>`

Node.js Equivalent	`qb.parseBindings(`
	`  qb.value(`
	`  qb.element( qb.qname(`
	`  'http://marklogic.com/entity-services','title')),`
	`  qb.bind('entity-type')))`

If your entity type definition assigns properties to range indexes or word lexicons, the generated options will include additional named constraints. You can define similar parse bindings for these constraints. For more details, see Using Constraints in a String Query in the *Node.js Application Developer's Guide*.

Use `qb.slice(qb.extract...))` to mimic the behavior of the `extract-document-data` option. This causes the search to return just the matched entity instance, instead of the entire envelope document. This is the section of the query that required us to define a namespace prefix binding for "es".

Description	Example

Generated Option	`<search:extract-document-data selected="include">`
	`  <search:extract-path xmlns:es="http://marklogic.com/entity-services">`
	`  //es:instance/(Person)`
	`  </search:extract-path>`
	`</search:extract-document-data>`

Node.js Equivalent	`qb.where(...)`
	`  .slice(qb.extract({`
	`  paths: ['//es:instance/(Person)'],`
	`  selected: 'include'`
	`  }))`

6.9.3 REST Client API

The REST Client API enables client applications to interact with MarkLogic using HTTP requests. The API enables you to search documents using a variety of query styles, including string query, structured query, QBE, and combined query. For more details, see Using and Configuring Query Features in the *REST Application Developer's Guide*.

Recall that you can generate Search API compatible query options using the Entity Services API; for details, see "Generating Query Options for Searching Instances" on page 141. To take advantage of these option, you must either pre-install the options as described in "Pre-Installing Query Options" on page 173, or embed them in a combined query.

The following command uses a string query to find all `Person` envelope documents where the instance data contains the word "washington". The command uses a string query and assumes the options are pre-installed under the name "person-1.0.0". The search is performed by a request to `GET:/v1/search`.

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X GET -i \
    'http://localhost:8000/LATEST/search?q=entity-type:person AND
washington&options=person-1.0.0&database=es-gs'
```

If you run the command against the model and instance data from "Getting Started With Entity Services" on page 19, the request returns the entity instance data for "Martha Washington" and "George Washington" in the `<search:extracted/>` element of the response. For example:

```
<search:response snippet-format="empty-snippet" total="2"
    start="1" page-length="10" selected="include"
    xmlns:search="http://marklogic.com/appservices/search">
  <search:result index="1" uri="/es-gs/env/2345.xml"
      path="fn:doc(&amp;quot;/es-gs/env/2345.xml&amp;quot;)"
      score="15872" confidence="0.4703847" fitness="0.7823406"
      href="/v1/documents?uri=%2Fes-gs%2Fenv%2F2345.xml&amp;database=es-ex"
      mimetype="application/xml" format="xml">
    <search:snippet/>
    <search:extracted kind="element">
      <Person>
        <id>2345</id>
        <firstName>Martha</firstName>
        <lastName>Washington</lastName>
        <fullName>Martha Washington</fullName>
      </Person>
    </search:extracted>
  </search:result>
  ...
  <search:qtext>entity-type:person AND washington</search:qtext>
  <search:metrics>...</search:metrics>
</search:response>
```

The response includes only the matched entity instances because of the `extract-document-data` option. For a discussion of the generated options used in this example, see "Example: Using the Search API for Instance Queries" on page 176.

You can use the request Accept headers to retrieve results as JSON, but the "extracted" property value in the JSON response will contain serialized XML because entity data is stored as XML in the envelope documents.

To perform an equivalent search without pre-installing the options use a combined query that embeds the options in a `<search:search/>` element. Use the combined query as the request body for `POST:/v1/search`. For example, create a combined query of the following form:

```
<search xmlns="http://marklogic.com/appservices/search">
  <qtext>entity-type:Person AND washington</qtext>
  <options> <!-- the generated options here -->
    ...
  </options>
</search>
```

For more details, see Specifying Dynamic Query Options with Combined Query in the *REST Application Developer's Guide*.

6.10 Example: Using SPARQL for Instance Queries

The default TDE template that you can generate with the Entity Services API auto-generates triples from your entity envelope documents, as long as the instance entity type defines a primary key.

> **Note:** You must install the template before this triple generation can occur. For details, see "Generating a TDE Template" on page 122.

The default generated triples express facts such as the following, where the instance is identified by primary key. For more details, see "Characteristics of a Generated Template" on page 124.

- This instance has this entity type. For example, this triple expresses the fact that an entity instance has the type defined by the IRI `<http://example.org/example-person/Person-1.0.0/Person>` The type IRI takes the form of {*baseURI*}{*modelTitle*}-{*modelVersion*}/{*entityTypeName*}.

  ```
  <http://example.org/example-person/Person-1.0.0/Person/1234>
    <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://example.org/example-person/Person-1.0.0/Person>
  ```

- This instance is defined by this envelope document (identified by URI). For example, the following triple expresses the fact that a particular entity instance is defined by the envelope document with URI `/es-gs/env/1234.xml`. The entity instance IRI takes the form of {*baseURI*}{*modelTitle*}-{*modelVersion*}/{*entityTypeName*}/{primaryKey}.

  ```
  <http://example.org/example-person/Person-1.0.0/Person/1234>
    <http://www.w3.org/2000/01/rdf-schema#isDefinedBy>
    "/es-gs/env/1234.xml"^^xs:anyURI
  ```

You can also extend the template to generate additional triples or manually add triples to the database.

The following SPARQL query returns the URIs of all `Person` entities.

```
prefix es: <http://marklogic.com/entity-services#>
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
prefix xs: <http://www.w3.org/2001/XMLSchema#>

select ?uri
where {
  ?person a ?personType .
  ?person rdfs:isDefinedBy ?docUri .
  ?personType es:title 'Person'
  bind(xs:string(?docUri) as ?uri)
}
```

If you generate and install a TDE template using the model from "Getting Started With Entity Services" on page 19, then the query display the following entity envelope document URIs:

```
/es-gs/env/1234.xml
/es-gs/env/2345.json
/es-gs/env/3456.xml
```

You can query facts about your instance data using the following APIs.

- The `sem:sparql` XQuery function or the `sem.sparql` Server-Side JavaScript functions.
- The Client APIs; see Client-Side APIs for Semantics in the *Semantics Developer's Guide*.
- The Optic API XQuery; see the `op:from-triples` XQuery function or the `op.fromTriples` JavaScript function.

6.11 Example: Using SQL for Instance Queries

If you generate and install a TDE template for your model, then MarkLogic auto-generates row data from your entity envelope documents. The row data enables you to query your entity instances as rows.

You must install the template before this row generation can occur. For details, see "Generating a TDE Template" on page 122. To learn more about the characteristics of the row data, see "Characteristics of a Generated Template" on page 124.

You can evaluate SQL using the `xdmp:sql` XQuery function or the `xdmp.sql` Server-Side JavaScript function, as shown below. You can also use the Optic API to query row data; see "Example: Using the Optic API for Instance Queries" on page 191.

The following example finds all Person rows where the "lastName" column has the value "Washington" and returns the value of the "fullName" column for the matched rows.

Language	Example

XQuery	`xquery version "1.0-ml";`
	`xdmp:sql("`
	`  SELECT Person.fullName`
	`  FROM Person`
	`  WHERE Person.lastName='Washington'`
	`", "format")`

JavaScript	`xdmp.sql(`
	`  'SELECT Person.fullName ' +`
	`  'FROM Person ' +`
	`  'WHERE Person.lastName=\'Washington\''`
	`, "format");`

If you generate and install a TDE template from the model from "Getting Started With Entity Services" on page 19 and run the query against the instance data, then you should see output similar to the following:

```
| Person.Person.fullName|
| Martha Washington|
| George Washington|
```

6.12 Example: Using the Optic API for Instance Queries

If you generate and install a TDE template for your model, then MarkLogic auto-generates row data from your entity envelope documents. The row data enables you to query your entity instances as rows. If an entity defines a primary key, the template also causes MarkLogic to auto-generate semantic triples about each instance.

> **Note:** You must install the template before this auto-generation can occur. For details, see "Generating a TDE Template" on page 122.

The examples in this section are based on the model and instance data from "Getting Started With Entity Services" on page 19. The examples also assume you have generated and installed a template based on this model, as shown in "Generating a TDE Template" on page 122.

- Querying Triples Using the Optic API
- Querying Rows Using the Optic API

6.12.0.1 Querying Triples Using the Optic API

This example uses the Optic API to query semantic "facts" about instance data. You can also use the Optic API for semantic queries on an entity model. For examples using SPARQL, see "Example: Using SPARQL for Instance Queries" on page 189 and "Example: Using SPARQL for Model Queries" on page 174.

The following example finds all entity instances that have Person type.

Language	Example

XQuery	`import module namespace op =`
	`  "http://marklogic.com/optic" at "/MarkLogic/optic.xqy";`
	
	`let $ps :=`
	`  op:prefixer("http://example.org/example-person/Person-1.0.0/")`
	`let $rdf :=`
	`  op:prefixer("http://www.w3.org/1999/02/22-rdf-syntax-ns#")`
	`return`
	`  op:from-triples( ( op:pattern( op:col("instanceIri"),`
	`  $rdf("type"),`
	`  op:col("type") ) ) )`
	`  =>op:where( op:eq( op:col("type"), $ps("Person")))`
	`  =>op:result()`

JavaScript	`const op = require('/MarkLogic/optic');`
	`const ps =`
	`  op.prefixer('http://example.org/example-person/Person-1.0.0/');`
	`const rdf =`
	`  op.prefixer('http://www.w3.org/1999/02/22-rdf-syntax-ns#');`
	
	`op.fromTriples( op.pattern( op.col('instanceIri'),`
	`  rdf('type'),`
	`  op.col('type')) )`
	`  .where(op.eq(op.col('type'), ps('Person')))`
	`  .result();`

If you run the query in Query Console against the expected configuration, it matches the following instance IRIs:

```
http://example.org/example-person/Person-1.0.0/Person/1234
http://example.org/example-person/Person-1.0.0/Person/2345
http://example.org/example-person/Person-1.0.0/Person/3456
```

6.12.0.2 Querying Rows Using the Optic API

This example uses the Optic API to query instance data as rows. For examples using SQL, see “Example: Using SQL for Instance Queries” on page 190.

The following query finds the Person entity with an id property of “2345”. Each entity instance is represented by a row in the Person table, with a column for each property.

Language	Example

XQuery	`import module namespace op =`
	`  "http://marklogic.com/optic" at "/MarkLogic/optic.xqy";`
	`import module namespace opxs =`
	`  "http://marklogic.com/optic/expression/xs"`
	`  at "/MarkLogic/optic/optic-xs.xqy";`
	
	`op:from-view("Person", "Person")`
	`  =>op:where(op:eq(op:col("id"), opxs:string("2345")))`
	`  =>op:select( (op:col("fullName")) )`
	`  =>op:result()`

JavaScript	`var op = require("/MarkLogic/optic");`
	`var opxs = op.xs;`
	
	`op.fromView("Person", "Person")`
	`.where(op.eq(op.col("id"), opxs.string("2345")))`
	`.select( [op.col("fullName") ] )`
	`.result();`

If you run the query in Query Console against the expected configuration, it returns “Martha Washington”.

6.13 Where to Find Additional Information

You can find more examples in the Entity Services GitHub repository. For details, see “Exploring the Entity Services Open-Source Examples” on page 15.

For more details on the APIs used in this chapter, see the following resources:

- The *Search Developer’s Guide*
- Searching in the *Java Application Developer’s Guide*
- Querying Documents and Metadata in the *Node.js Application Developer’s Guide*
- Using and Configuring Query Features in the *REST Application Developer’s Guide*
- The *SQL Data Modeling Guide*
- Optic API for Multi-Model Data Access in the *Application Developer’s Guide*
- Semantic Queries in the *Semantics Developer’s Guide*

7.0 Technical Support

MarkLogic provides technical support according to the terms detailed in your Software License Agreement or End User License Agreement.

We invite you to visit our support website at http://help.marklogic.com to access information on known and fixed issues, knowledge base articles, and more. For licensed customers with an active maintenance contract, see the Support Handbook for instructions on registering support contacts and on working with the MarkLogic Technical Support team.

Complete product documentation, the latest product release downloads, and other useful information is available for all developers at http://developer.marklogic.com. For technical questions, we encourage you to ask your question on Stack Overflow.

8.0 Copyright

MarkLogic Server 9.0 and supporting products.
Last updated: August 5, 2020

MarkLogic technology is protected by one or more U.S. Patent Nos. 7,127,469, 7,171,404, 7,756,858, 7,962,474, 8,935,267, 8,892,599, 9,092,507, 10,108,742, 10,114,975, 10,311,088, 10,325,106, 10,339,337, 10,394,889, and 10,503,780.

MarkLogic software incorporates certain third-party software under license. Third-party attributions, copyright notices, and other disclosures required under license are available in the respective notice document for your version of the MarkLogic software.