Elide英文文档

目　录

致谢

介绍

chat on gitter

build passing

maven central 4.2.4

coverage 72%

Elide简介

Elide是一个互联网和移动端应用数据API搭建平台，只需要一个简单的JPA注释模型
就能帮您轻松搭建GraphQL和JSON API web 服务的。

标准完善的数据安全保障

Elide提供极具规则性，简单易懂的语法规则，让您轻松搞定实体（entity）的安全访问。

移动端性能优化 API

JSON-API和GraphQL能够帮助开发者仅通过一次API接口访问就能获取与某个实体相关的所有数据，而且在移动端传输过程中减少所有不
必要的数据，只返回被请求的数据部分。我们为您精心设计的数据处理系统帮助您解决很多常见的数据应用开发问题，例如：

- 在单次请求操作中实现创建实体数据，同时将其加入现有的实体库
- 创建存储关系复杂的多个相关实体（实体关联图），并将它们并入现有的实体库
- 你可以选择完全删除某个实体数据，也可以选择接触实体关联（并不删除任何数据）
- 完全自由修改实体关联定义
- 修改实体数据的同时还可以访问加入新建实体

Elide还完全支持数据筛选，排序，分页。

任何数据写入都可以保证原子性

无论是JSON-API还是GraphQL，Elide支持单个请求中实现多个数据模型的修改操作。创建新的实体，添加实体关系，修改和删除实体保证
事务的原子性（Atomicity）。

支持自定义数据持久化机制

你可以用Elide自定义您的持久化方法策略。您可以使用Elide默认支持的ORM或者使用自行开发的数据存储机制。

数据模型一览无余

您可以借助自动生成的Swagger文档或者GraphQL数据模型了解，学习，和编写Elide API查询语句。

配置轻松自由

您可以按照您的意思轻松配置您需要的数据模型操作，比如添加复杂的二次运算实体数据，数据检查标注（data validation
annotations），或者是自动以的访问请求中间链模块。

文档

更多使用指南，请参见elide.io.

开发

如果您想给Elide贡献代码，请在IDE中添加Lombok插件。

教程（英文）

Create a JSON API REST Service With Spring Boot and Elide

Custom Security With a Spring Boot/Elide Json API Server

Logging Into a Spring Boot/Elide JSON API Server

Securing a JSON API REST Service With Spring Boot and Elide

Creating Entities in a Spring Boot/Elide JSON API Server

Updating and Deleting with a Spring Boot/Elide JSON API Server

开源许可

Getting Started

Getting Started

So You Want An API?

The easiest way to get started with Elide is to use the elide-standalone library. The standalone library bundles all ofthe dependencies you will need to stand up a web service. This tutorial will use elide-standalone, all of the code isavailable here–if you want to see a more fully fleshed out example of the standalone library checkout thisKotlyn blog example.

Create A Bean

JPA beans are some of the most important code in any Elide project. Your beans are the view of your data model that youwish to expose. In this example we will be modeling a software artifact repository since most developers have ahigh-level familiarity with artifact repositories such as Maven, Artifactory, npm, and the like. If you are interested,the code is tagged for each step so you can follow along.

The first bean we'll need is the `ArtifactGroup` bean, for brevity we will omit package names and import statements. Thiswill represent the `<groupId>` in Maven's dependency coordinates.

```java
@Include(rootLevel = true)
@Entity
public class ArtifactGroup {
    @Id
    public String name = "";
}
```

Spin up the API

So now we have a bean, but without an API it is not do very useful. Before we add the API component of the we need tocreate the schema in the database that our beans will use. Download and run the demo setup script; thisdemo uses MySQL, feel free to modify the setup script if you are using a different database provider.

You may notice that there are more tables that just `ArtifactGroup` , and that the `ArtifactGroup` table has more fieldsthat our bean. Not only will our bean work just fine, we expect that beans will normally expose only a subset of thefields present in the database. Elide is an ideal tool for building micro-services, each service in your system canexpose only the slice of the database that it requires.

Classes

Bringing life to our API is trivially easy. We need two new classes: Main and Settings.

- Main.java
- Settings.java

```java
public class Main {
    public static void main(String[] args) throws Exception {
        ElideStandalone app = new ElideStandalone(new Settings());
        app.start();
    }
}
```

```java
public class Settings implements ElideStandaloneSettings {
    @Override
    public String getModelPackageName() {
        return ArtifactGroup.class.getPackage().getName();
    }
}
```

Supporting Files

Elide standalone also requires a hibernate config file. By default the standalone API expects to find your hibernateconfig at `./settings/hibernate.cfg.xml` (with respect to CWD when you launch the app). If you want tosee the logs from your shiny new API you will also want a logback config. Your logback config should goin `src/main/resources` so logback can find it.

Running

With these new classes you have two options for running your project, you can either run the `Main` class using yourfavorite IDE, or we can add the following snippet to our gradle build script and run our project with ./gradlew run

```
plugins {
  ...
  id 'application'
}

mainClassName = 'com.example.repository.Main' // the actual path to your Main class should go here
```

With the `Main` and `Settings` classes we can now run our API. If you navigate to `http://localhost:8080/api/v1/artifactGroup` in your browser you can see some of the sample data that the bootstrapscript added for us. Exciting!

```
{"data":[{"type":"artifactGroup","id":"com.example.repository"},
{"type":"artifactGroup","id":"com.yahoo.elide"}]}
```

Adding More Data

Now that we have an API that returns data, let’s add some more interesting behavior. Let’s update `ArtifactGroup` , andadd the `ArtifactProduct` and `ArtifactVersion` classes–which will be the `<artifactId>` and `<version>` tagsrespectively.

- ArtifactGroup.java
- ArtifactProduct.java
- ArtifactVersion.java

```
@Include(rootLevel = true)
@Entity
public class ArtifactGroup {
    @Id
    public String name = "";

    public String commonName = "";

    public String description = "";

    @OneToMany(mappedBy = "group")
    public List<ArtifactProduct> products = new ArrayList<>();
}
```

```
@Include
@Entity
public class ArtifactProduct {
    @Id
    public String name = "";

    public String commonName = "";

    public String description = "";

    @ManyToOne
    public ArtifactGroup group = null;

    @OneToMany(mappedBy = "artifact")
    public List<ArtifactVersion> versions = new ArrayList<>();
}
```

```
@Include
@Entity
public class ArtifactVersion {
    @Id
    public String name = "";

    public Date createdAt = new Date();

    @ManyToOne
    public ArtifactProduct artifact;
}
```

We add the missing fields to `ArtifactGroup` since we anticipate user will want to add some informative metadata to helpusers find the products and artifacts they are interested in. If we restart the API and request `/artifactGroup` we'llsee the other metadata we just added.

```
{"data":[{"type":"artifactGroup","id":"com.example.repository","attributes":{"commonName":"Example Repository","description":"The code for this project"},"relationships":{"products":{"data":[]}}},{"type":"artifactGroup","id":"com.yahoo.elide","attributes":{"commonName":"Elide","description":"The magical library powering this project"},"relationships":{"products":{"data":[{"type":"artifactProduct","id":"elide-core"},{"type":"artifactProduct","id":"elide-standalone"},{"type":"artifactProduct","id":"elide-datastore-hibernate5"}]}}}]}
```

So now we have an API that can display information for a full `<group>:<product>:<version>` set. We can fetch data fromour API in the following ways:

```
List groups:                /artifactGroup/
Show a group:               /artifactGroup/<group id>
List a group's products:    /artifactGroup/<group id>/products/
Show a product:             /artifactGroup/<group id>/products/<product id>
List a product's versions:  /artifactGroup/<group id>/products/<product id>/versions/
Show a version:             /artifactGroup/<group id>/products/<product id>/versions/<version id>
```

We can now fetch almost all of the data we would wish, but let's clean it up a bit. Right now all of our data types areprefixed with Artifact. This might make sense in Java so that we don't have naming collisions with classes from otherlibraries, however the consumers of our API do not care about naming collisions. We can control how Elide exposes ourclasses by setting the type on our `@Include` annotations.

```
@Include(type = "group")
@Entity
public class ArtifactGroup { ... }

@Include(type = "product")
@Entity
public class ArtifactProduct { ... }

@Include(type = "version")
@Entity
public class ArtifactVersion{ ... }
```

Now, instead of making a call to `http://localhost:8080/api/v1/artifactGroup` to fetch our data, we make a request to `http://localhost:8080/api/v1/group` . Our API returns the same data as before, mostly. The types of our objects nowreflect our preferences from the `Include` annotations.

```
{"data":[{"type":"group","id":"com.example.repository",...},
{"type":"group","id":"com.yahoo.elide",..."relationships":{"products":{"data":[{"type":"product","id":"elide-core"},...]}}}]}
```

Writing Data

So far we have defined our views on the database and exposed those views over HTTP. This is great progress, but so farwe have only read data from the database.

Inserting Data

Fortunately for us adding data is just as easy as reading data. For now let’s use cURL to put data in the database.

```
curl -X POST http://localhost:8080/api/v1/group/com.example.repository/products \
  -H"Content-Type: application/vnd.api+json" -H"Accept: application/vnd.api+json" \
  -d '{"data": {"type": "product", "id": "elide-demo"}}'
```

When you run that cURL call you should see a bunch of json returned, that is our newly inserted object! If we query `http://localhost:8080/api/v1/group/com.example.repository/products/`

```
{"data":[{"type":"product","id":"elide-demo","attributes":{"commonName":"","description":""},"relationships":
{"group":{"data":{"type":"group","id":"com.example.repository"}},"versions":{"data":[]}}}]}
```

Modifying Data

Notice that, when we created it, we did not set any of the attributes of our new product record. Unfortunately for ourusers this leaves the meaning of our elide-demo product ambiguous. What does it do, why should they use it? Updating ourdata to help our users is just as easy as it is to add new data. Let’s update our bean with the following cURL call.

```
curl -X PATCH http://localhost:8080/api/v1/group/com.example.repository/products/elide-demo \
  -H"Content-Type: application/vnd.api+json" -H"Accept: application/vnd.api+json" \
  -d '{
    "data": {
      "type": "product",
      "id": "elide-demo",
      "attributes": {
        "commonName": "demo application",
        "description": "An example implementation of an Elide web service that showcases many Elide features"
      }
    }
  }'
```

It’s just that easy to create and update data using Elide.

```

```

> 原文: *http://elide.io/pages/guide/01-start.html*

Data Models

Data Models

Elide generates its API entirely based on the concept of **Data Models**. In summary, these are JPA-annotated Java classes that describe the *schema* for each exposed endpoint. While the JPA annotations provide a high-level description on how relationships and attributes are modeled, Elide provides a set of security annotations to secure this model. Data models are intended to be a *view* on top of the datastore or the set of datastores which support your Elide-based service.

NOTE: This page is a description on how to *create* data models in the backend using Elide. For more information on *interacting* with an Elide API, please see our API usage documentation.

JPA Annotations

The JPA (Java Persistence API) library provides a set of annotations for describing relationships between entities. Elide makes use of the following JPA annotations: `@Entity`, `@OneToOne`, `@OneToMany`, `@ManyToOne`, and `@ManyToMany`. Any JPA property or field that is exposed via Elide and is not a *relationship* is considered an *attribute* of the entity.

If you need more information about JPA, please review their documentation or see our examples below.

Exposing a Model as an Elide Endpoint

After creating a proper data model, exposing it through Elide requires you configure *include* it in Elide. Elide generates its API as a *graph*; this graph can only be traversed starting at a *root* node. Rootable entities are denoted by applying `@Include(rootLevel=true)` to the top-level of the class. Non-rootable entities can be accessed only as relationships through the graph.

```
@Include(rootLevel=true)
@Entity
public class Author {
    private Long id;
    private String name;
    private Set<Book> books;

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @ManyToMany
    public Set<Book> getBooks() {
        return books;
    }

    public void setBooks(Set<Book> books) {
        this.books = books;
    }
}
```

```
@Include
@Entity
public class Book {
    private Long id;
    private String title;
    private Set<Author> authors;

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    @ManyToMany
```

```java
    public Set<Author> getAuthors() {
        return authors;
    }

    public void setAuthors(Set<Author> authors) {
        this.authors = authors;
    }
}
```

Considering the example above, we have a full data model that exposes a specific graph. Namely, a root node of the type `Author` and a bi-directional relationship from `Author` to `Book`. That is, one can access all `Author` objects directly, but must go *through* an author to see information about any specific `Book` object.

All public getters and setters are exposed through the Elide API if they are not explicitly marked `@Transient` or `@Exclude`. `@Transient` allows a field to be ignored by both Elide and an underlying persistence store while `@Exclude` allows a field to exist in the underlying JPA persistence layer without exposing it through the Elide API.

Much of the Elide per-model configuration is done via annotations. For a description of all Elide-supported annotations, please check out the annotation overview.

Computed Attributes

A computed attribute is an entity attribute whose value is computed in code rather than fetched from a data store.

Elide supports computed properties by way of the `@ComputedAttribute` and `@ComputedRelationship` annotations. These are useful if your datastore is also tied to your Elide view data model. For instance, if you mark a field `@Transient`, a datastore such as Hibernate will ignore it. In the absence of the `@Computed*` attributes, Elide will too. However, when applying a computed property attribute, Elide will expose this field anyway.

A computed attribute can perform arbitrary computation and is exposed through Elide as a typical attribute. In the case below, this will create an attribute called `myComputedAttribute`.

```java
@Include
@Entity
public class Book {
    ...
    @Transient
    @ComputedAttribute
    public String getMyComputedAttribute(RequestScope requestScope) {
        return "My special string stored only in the JVM!";
    }
    ...
}
```

The same principles are analogous to `@ComputedRelationship` s.

Lifecycle Hooks

Lifecycle event triggers allow custom business logic (defined in functions) to be invoked during CRUD operations at three distinct phases:

- *Pre Security* - Executed prior to Elide *commit* security check evaluation.
- *Pre Commit* - Executed immediately prior to transaction commit but after all security checks have been evaluated.

- *Post Commit* - Executed immediately after transaction commit.
  There are two mechanisms to enable lifecycle hooks:

- The simplest mechanism embeds the lifecycle hook as methods within the entity bean itself. The methods are marked with @On… annotations (see below).

- Lifecycle hook functions can also be registered with the EntityDictionary when initializing Elide.

Annotation Based Hooks

There are separate annotations for each CRUD operation (*read*, *update*, *create*, and *delete*) and also each life cycle phase of the current transaction:

```
@Entity
class Book {
   @Column
   public String title;

   @OnReadPreSecurity("title")
   public void onReadTitle() {
      // title attribute about to be read but 'commit' security checks not yet executed.
   }

   @OnUpdatePreSecurity("title")
   public void onUpdateTitle() {
      // title attribute updated but 'commit' security checks not yet executed.
   }

   @OnUpdatePostCommit("title")
   public void onCommitTitle() {
      // title attribute updated & committed
   }

   @OnCreatePostCommit
   public void onCommitBook() {
      // book entity created & committed
   }
```

```

    /**
     * Trigger functions can optionally accept a RequestScope to access the user principal.
     */
    @OnDeletePreCommit
    public void onDeleteBook(RequestScope scope) {
       // book entity deleted but not yet committed
    }
}
```

All trigger functions can either take zero parameters or a single `RequestScope` parameter.

The `RequestScope` can be used to access the user principal object that initiated the request:

```
    @OnReadPostCommit("title")
    public void onReadTitle(RequestScope scope) {
       User principal = scope.getUser();

       //Do something with the principal object...
    }

```

Update trigger functions on fields can also take both a `RequestScope` parameter and a `ChangeSpec` parameter.The `ChangeSpec` can be used to access the before & after values for a given field change:

```
    @OnUpdatePreSecurity("title")
    public void onUpdateTitle(RequestScope scope, ChangeSpec changeSpec) {
       //Do something with changeSpec.getModified or changeSpec.getOriginal
    }

```

Specifying an annotation without a value executes the denoted method on every instance of that action (i.e. every update, read, etc.). However, if a value is specified in the annotation, then that particular method is only executed when the specific operation occurs to the particular field. Below is a description of each of these annotations and their function:

- @OnCreatePreSecurity This annotation executes immediately when the object is created, with fields populated, on the server-side after User checks but before *commit* security checks execute and before it is committed/persisted in the backend. Any non-user *inline* and *operation* CreatePermission checks are effectively *commit* security checks.
- @OnCreatePreCommit This annotation executes after the object is created and all security checks are evaluated on the server-side but before it is committed/persisted in the backend.
- @OnCreatePostCommit This annotation executes after the object is created and

committed/persisted on the backend.

- @OnDeletePreSecurity This annotation executes immediately when the object is deleted on the server-side but before *commit* security checks execute and before it is committed/persisted in the backend.
- @OnDeletePreCommit This annotation executes after the object is deleted and all security checks are evaluated on the server-side but before it is committed/persisted in the backend.
- @OnDeletePostCommit This annotation executes after the object is deleted and committed/persisted on the backend.
- @OnUpdatePreSecurity(value) If value is **not** specified, then this annotation executes on every update action to the object. However, if value is set, then the annotated method only executes when the field corresponding to the name in value is updated. This annotation executes immediately when the field is updated on the server-side but before *commit* security checks execute and before it is committed/persisted in the backend.
- @OnUpdatePreCommit(value) If value is **not** specified, then this annotation executes on every update action to the object. However, if value is set, then the annotated method only executes when the field corresponding to the name in value is updated. This annotation executes after the object is updated and all security checks are evaluated on the server-side but before it is committed/persisted in the backend.
- @OnUpdatePostCommit(value) If value is **not** specified, then this annotation executes on every update action to the object. However, if value is set, then the annotated method only executes when the field corresponding to the name in value is updated. This annotation executes after the object is updated and committed/persisted on the backend.
- @OnReadPreSecurity(value) If value is **not** specified, then this annotation executes every time an object field is read from the datastore. However, if value is set, then the annotated method only executes when the field corresponding to the name in value is read. This annotation executes immediately when the object is read on the server-side but before *commit* security checks execute and before the transaction commits.
- @OnReadPreCommit(value) If value is **not** specified, then this annotation executes every time an object field is read from the datastore. However, if value is set, then the annotated method only executes when the field corresponding to the name in value is read. This annotation executes after the object is read and all security checks are evaluated on the server-side but before the transaction commits.
- @OnReadPostCommit(value) If value is **not** specified, then this annotation executes every time an object field is read from the datastore. However, if value is set, then the annotated method only executes when the field corresponding to the name in value is read. This annotation executes after the object is read and the transaction commits.

Registered Function Hooks

To keep complex business logic separated from the data model, it is also possible to register `LifeCycleHook` functions during Elide initialization (since Elide 4.1.0):

```java
/**
 * Function which will be invoked for Elide lifecycle triggers
 * @param <T> The elide entity type associated with this callback.
 */
@FunctionalInterface
public interface LifeCycleHook<T> {
    /**
     * Run for a lifecycle event
     * @param elideEntity The entity that triggered the event
     * @param requestScope The request scope
     * @param changes Optionally, the changes that were made to the entity
     */
    public abstract void execute(T elideEntity,
                                 RequestScope requestScope,
                                 Optional<ChangeSpec> changes);
```

The hook functions are registered with the `EntityDictionary` by specifying the corresponding life cycle annotation (which defines when the hook triggers) alongwith the entity model class and callback function:

```java
//Register a lifecycle hook for deletes on the model Book
dictionary.bindTrigger(Book.class, OnDeletePreSecurity.class, callback);

//Register a lifecycle hook for updates on the Book model's title attribute
dictionary.bindTrigger(Book.class, OnUpdatePostCommit.class, "title", callback);
```

Initializers

Sometimes models require additional information from the surrounding system to be useful. Since all model objects in Elide are ultimately constructed by the `DataStore`, and because Elide does not directly depend on any specific dependency injection framework (though you can still use your own dependency injection frameworks), Elide provides an alternate way to initialize a model.

Elide can be configured with an `Initializer` implementation for a particular model class. An `Initializer` is any class which implements the following interface:

```java
@FunctionalInterface
public interface Initializer<T> {
    /**
     * Initialize an entity bean
     *
     * @param entity Entity bean to initialize
     */
    public void initialize(T entity);
}
```

Initializers can be configured in a custom `DataStore` when the method `populateEntityDictionary` is invoked:

```java
    public void populateEntityDictionary(EntityDictionary dictionary) {

        /* Assuming this DataStore extends another... */
        super.populateEntityDictionary(dictionary);

        /*
         * Create an initializer for model Foobar, passing any runtime configuration to
         * the constructor of the initializer.
         */
        ...

        /* Bind the initializer to Foobar.class */
        dictionary.bindInitializer(foobarInitializer, Foobar.class);
    }
```

Dependency Injection

Dependency injection in Elide can be achieved by using initializers. To do so, implement your own store (or extend an existing store) and implement something like the following:

```java
public class MyStore extends HibernateStore {
    private final Injector injector;

    public MyStore(Injector injector, ...) {
        super(...);
        this.injector = injector;
    }

    @Override
    public void populateEntityDictionary(EntityDictionary) {
        /* bind your entities */
        for (Class<?> entityClass : yourEntityList) {
            dictionary.bindInitializer(injector::inject, entityClass);
        }
    }
}
```

Ultimately, each time an object of `entityClass` type is instantiated by Elide, Elide will run an initializer that allows the injection framework to inject into the new object.

If you’re using the `elide-standalone` artifact, then this is already done by default.

Validation

Data models can be validated using bean validation. This requires*JSR303* data model annotations and wiring in a bean validator in the `DataStore` .

Philosophy

Data models are intended to be a *view* on top of the datastore or the set of datastores which support your Elide-based service. While other JPA-based workflows often encourage writing data models that exactly match the underlying schema of the datastore, we propose a strategy of isolation on per-service basis. Namely, we recommend creating a data model that only supports precisely the bits of data you need from your underlying schema. Often times there will be no distinction when first building your systems. However, as your systems scale and you develop multiple services with overlapping datastore requirements, isolation often serves as an effective tool to **reduce interdependency** among services and **maximize the separation of concern**. Overall, while models can correspond to your underlying datastore schema as a one-to-one representation, it’s not always strictly necessary and sometimes even undesireable.

As an example, let’s consider a situation where you have two Elide-based microservices: one for your application backend and another for authentication (suppose account creation is performed out-of-band for this example). Assuming both of these rely on a common datastore, they’ll both likely want to recognize the same underlying *User* table. However, it’s quite likely that the authentication service will only ever require information about user **credentials** and the application service will likely only ever need user **metadata**. More concretely, you could have a system that looks like the following:

Table schema:

```
id
userName
password
firstName
lastName
```

Authentication schema:

```
id
userName
password
```

Application schema:

```
id
```

```
userName
firstName
lastName
```

While you could certainly just use the raw table schema directly (represented as a JPA-annotated data model) and reuse it across services, the point is that you may be over-exposing information in areas where you may not want to. In the case of the *User* object, it's quite apparent that the application service should never be *capable* of accidentally exposing a user's private credentials. By creating isolated views per-service on top of common datastores, you sacrifice a small bit of DRY principles for much better isolation and a more targeted service. Likewise, if the underlying table schema is updated with a new field that neither one of these services needs, neither service requires a rebuild and redeploy since the change is irrelevant to their function.

A note about microservices: Another common technique to building microservices is for each service to have its own set of datastores entirely independent from other services (i.e. no shared overlap); these datastores are then synced by other services as necessary through a messaging bus. If your system architecture calls for such a model, it's quite likely you will follow the same pattern we have outlined here with *one key difference*: the underlying table schema for your *individual service's datastore* will likely be exactly the same as your service's model representing it. However, overall, the net effect is the same since only the relevant information delivered over the bus is stored in your service's schema. In fact, this model is arguably more robust in the sense that if one datastore fails not all services necessarily fail.

> 原文: *http://elide.io/pages/guide/02-data-model.html*

Security

Security

Core Concepts

API authentication is largely a solved problem and generally outside the scope of Elide.Authorization - the act of verifying data and operation access for an *already authenticated user* in the Elide framework involves a few core concepts:

- **User** - Each API request is associated with a user principal. The user is opaque to the Elide framework but is passed to developer-defined *check* functions that evaluate arbitrary logic or build filter expressions.
- **Checks** - a function *or* filter expression that grants or denies a user **permission** to perform a particular action.
- **Permissions** - a set of annotations (read, update, delete, create, and share) that correspond to actions on the data model’s entities and fields. Each **permission** is decorated with one or more checks that are evaluated when a user attempts to perform that action.

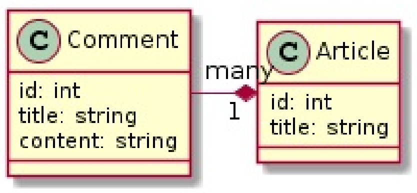
Security Evaluation

Security is applied hierarchically with three goals:

- **Granting or denying access.** When a model or field is accessed, a set of checks are evaluated to determine if the access will be denied (i.e. 403 HTTP status code (JSON-API) or GraphQL error object) or permitted. If a user has explicitly requested access to part of the data model they should not see, the request will be rejected.
- **Filtering Collections.** If a model has read permissions defined, these checks are evaluated against each model that is a member of the collection. Only the models the user has access to (by virtue of being of being able to read at least one of the model’s fields) are returned in the response.
- **Filtering a model.** If a user has read access to a model, but only for a subset of a model’s fields, the disallowed fields are excluded from the output (rather than denying the request). However, when the user explicitly requests a field-set that contains a restricted field, the request is rejected rather than filtered.

Hierarchical Security

Both JSON-API and GraphQL define mechanisms to fetch and manipulate entities defined by the data model schema. Some (rootable) entities can be reached directly by providing their data type and unique identifier in the query. Other entities can only be reached through relationships to other entities– by traversing the entity relationship graph. The Elide framework supports both methods of access. This is beneficial because it alleviates the need for all models to be accessible at the root of the graph. When everything is exposed at the root, the developer needs to enumerate all of the valid access patterns for all of the data models which quickly becomes unmanageable. In addition to eliminating redundancy in security declaration, this form of security can have significant performance benefits for enforcing security on large collections stored in key-value stores that have limited ability for the underlying persistence layer to directly apply security filters. It is often possible to deny access to an entire collection (i.e. hierarchical relationship) before attempting to verify access to each individual member within that collection. Typically, security rules only need to be defined for a subset of models and relationships– often near the roots of the graph. Applying security rules to the relationships to prune the graph can eliminate invalid access patterns. To better understand the sequence of how security is applied, consider the data model depicted in Figure 1 consisting of articles where each contains zero or more comments.



The request to update a specific comment of a particular article involves the following permission checks:

- Read permission check on the Article’s comments field.
- Update permission check on the Comment’s title field.
  When a client modifies one side of a bidirectional relationship, Elide will automatically update the opposite side of the relationship. This implies that the client must have permission to write both sides of the relationship.

Checks

Checks are simply functions that either return:

- whether or not access should be granted to the requesting user.
- a filter expression that can be used to filter a collection to what is visible to a given user.
  Checks can either be invoked:
- immediately prior to the (create, read, update, and delete) action being performed.
- immediately before committing the transaction that wraps the entire API request.

> The former is useful for checks that depend on data that is already persisted. Read, delete, and most update checks fall into this category. The latter is useful for checks involving complex mutations to the object graph for newly created data. These checks cannot execute until the data model has been fully updated to reflect the entire set of modifications. Corresponding to these categories, there are two types of checks in Elide: InlineCheck and CommitCheck. Checks must be implemented by extending one of these abstract classes. The class you choose to extend impacts how Elide will evaluate your the check. There are specific types of check described by the following hierarchy:

`InlineCheck` is the abstract super class of the three specific variants:

Operation Checks

Operation checks are inline checks whose evaluation requires the entity instance being read from or written to. They operate in memory of the process executing the Elide library.

Operation checks are expected to implement the following `Check` interface:

```
/**
 * Determines whether the user can access the resource.
 *
 * @param object Fully modified object
 * @param requestScope Request scope object
 * @param changeSpec Summary of modifications
 * @return true if security check passed
 */
boolean ok(T object, RequestScope requestScope, Optional<ChangeSpec> changeSpec);
```

User Checks

User checks depend strictly on the user principal. These are inline checks (i.e. they run as operations occur rather than deferring until commit time) and only take a User object as input. Because these checks only depend on who is performing the operation and not on what has changed, these checks are only evaluated once per request - an optimization that accelerates the filtering of large collections.

User checks are expected to implement the following `Check` interface:

```
/**
 * Method reserved for user checks.
 *
 * @param user User to check
 * @return True if user check passes, false otherwise
 */
boolean ok(User user);
```

Filter Expression Checks

In some cases, the check logic can be pushed down to the data store itself. For example, a filter can be added to a database query to remove elements from a collection where access is disallowed. These checks return a `FilterExpression` predicate that your data store can use to limit the queries that it uses to marshal the data. Checks which extend the `FilterExpressionCheck` must conform to the interface:

```

/**
 * Check for FilterExpression. This is a super class for user defined FilterExpression check. The subclass should
 * override getFilterExpression function and return a FilterExpression which will be passed down to datastore.
 * @param <T> Type of class
 */
public abstract class FilterExpressionCheck<T> extends InlineCheck<T> {

    /**
     * Returns a FilterExpression from FilterExpressionCheck.
     * @param entityClass The entity collection to filter
     * @param requestScope Request scope object
     * @return FilterExpression for FilterExpressionCheck.
     */
    public abstract FilterExpression getFilterExpression(Class<?> entityClass, RequestScope requestScope);
}
```

Most `FilterExpressionCheck` functions construct a `FilterPredicate` which is a concrete implementation of the `FilterExpression` interface:

```
/**
 * Constructs a filter predicate
 * @param path The path through the entity relationship graph to a particular attribute to filter on.
 * @param op The filter comparison operator to evaluate.
 * @param values The list of values to compare the attribute against.
 */
public FilterPredicate(Path path, Operator op, List<Object> values) {
 ...
}
```

Here is an example to filter the Author model by book titles:

```
   //Construct a filter for the Author model for books.title == 'Harry Potter'
   Path.PathElement authorPath = new Path.PathElement(Author.class, Book.class, "books");
   Path.PathElement bookPath = new Path.PathElement(Book.class, String.class, "title");
   List<Path.PathElement> pathList = Arrays.asList(authorPath, bookPath);
   Path path = new Path(pathList);

   return new FilterPredicate(path, Operator.IN, Collections.singletonList("Harry Potter"));
```

Filter expression checks are most important when a security rule is tied in some way to the data itself. By pushing the security rule down to the datastore, the data can be more efficiently queried which vastly improves performance. Moreover, this feature is critical for implementing a service that requires complex security rules (i.e. anything more than role-based access) on large collections.

User

Each request is associated with a user. The user is computed by a function that you provide conforming to the interface:

```
Function<SecurityContext, Object>
```

Permission Annotations

The permission annotations include `ReadPermission` , `UpdatePermission` , `CreatePermission` , `DeletePermission` , and `SharePermission` . Permissions are annotations which can be applied to a model at the `package` , `entity` , or `field` -level. The most specific annotation always take precedence (`package < entity < field`). More specifically, a field annotation overrides the behavior of an entity annotation. An entity annotation overrides the behavior of a package annotation. Entity annotations can be inherited from superclasses. When no annotation is provided at any level, access is implicitly granted for `ReadPermission` , `UpdatePermission` , `CreatePermission` , and `DeletePermission` and implicitly denied for `SharePermission` .

The permission annotations wrap a boolean expression composed of the check(s) to be evaluated combined with `AND` , `OR` , and `NOT` operators and grouped using parenthesis. The checks are uniquely identified within the expression by a string - typically a human readable phrase that describes the intent of the check (*“principal is admin at company OR principal is super user with write permissions”*). These strings are mapped to the explicit `Check` classes at runtime by registering them with Elide. When no registration is made, the checks can be identified by their fully qualified class names. The complete expression grammar can be found here.

To better understand how permissions work consider the following sample code. (Only the relevant portions are included.)

- User.java
- Post.java
- Comment.java
- IsOwner.java
- IsSuperuser.java
- MyElide.java

```
@ReadPermission(expression = "Prefab.Roll.All")
```

```java
@UpdatePermission(expression = "user is a superuser OR user is this user")
@DeletePermission(expression = "user is a superuser OR user is this user")
@SharePermission(expression = "Prefab.Role.All")
public class User {
    String name;

    @OneToMany(mappedBy = "author")
    Set<Post> posts;
}
```

```java
@ReadPermission(expression = "Post.isVisible OR User.ownsPost OR user is a superuser")
@UpdatePermission(expression = "user owns this post now")
@CreatePermission(expression = "user owns this post")
@DeletePermission(expression = "user owns this post now")
@SharePermission(expression = "Prefab.Role.All")
public class Post {
    @ManyToOne
    User author;

    @UpdatePermission(expression = "user owns this post now OR user is a superuser now")
    boolean published;

    @OneToMany(mappedBy = "post")
    Set<Comment> comments;
}
```

```java
// user has to be able to see the post and to see the comment, or else be a super user
@ReadPermission(expression = "((Post.isVisible OR User.ownsPost) AND (comment is visible OR user made this comment)) OR user is a superuser")
@UpdatePermission(expression = "user made this comment")
@CreatePermission(expression = "post is visible now")
@DeletePermission(expression = "user made this comment")
@SharePermission(expression = "Prefab.Role.All")
public class Comment {
    @ManyToOne
    User author;
    @ManyToOne
    Post post;
    @UpdatePermission(expression = "user owns this post now OR user is a superuser now")
    boolean suppressed;
}
```

```java
public class IsOwner {
    public static class Inline<Post> extends OperationCheck {
        @Override
        boolean ok(Post post, RequestScope requestScope, Optional<ChangeSpec> changeSpec) {
            return post.author.equals(requestScope.getUser());
        }
    }
    public static class AtCommit<Post> extends CommitCheck {
        @Override
        boolean ok(Post post, RequestScope requestScope, Optional<ChangeSpec> changeSpec) {
            return post.author.equals(requestScope.getUser());
        }
    }
}
```

```
public class IsSuperuser {
    public static class Inline<User> extends OperationCheck {
        @Override
        boolean ok(User user, RequestScope requestScope, Optional<ChangeSpec> changeSpec) {
            return user.isSuperuser();
        }
    }
}
```

```
HashMap checks = new HashMap();
checks.put("post is visible now", IsPublished.Inline.class);
checks.put("post is visible", IsPublished.AtCommit.class);
checks.put("user owns this post now", IsOwner.Inline.class);
checks.put("user owns this post", IsOwner.AtCommit.class);
checks.put("user is a superuser", IsSuperuser.Inline.class);
//...
EntityDictionary dictionary = new EntityDictionary(checks);
```

You will notice that `IsOwner` actually defines two check classes; it does so because we might want to evaluate the same logic at distinct points in processing the request (inline when reading a post and at commit when creating a post). For example, we could not apply `IsOwner.Inline` when creating a new post because the post's author has not yet been assigned. Once the post has been created and all fields assigned by Elide, the security check can be evaluated.

Contrast `IsOwner` to `IsSuperuser` which only defines one check. `IsSuperuser` only defines one check because it only depends on who is performing the action and not on the data model being manipulated.

Read

`ReadPermission` governs whether a model or field can be read by a particular user. If the expression evaluates to `true` then access is granted. Notably, `ReadPermission` is evaluated as the user navigates through the entity relationship graph. Elide's security model is focused on field-level access, with permission annotations applied on an entity or package being shorthand for applying that same security to every field in that scope. For example, if a request is made to `GET /users/1/posts/3/comments/99` the permission execution will be as follows:

- ReadPermission on User<1>#posts
- ReadPermission on Post<3>#comments
- ReadPermission on any field on Comment<99>

If all of these checks succeed, then the response will succeed. The contents of the response are determined by evaluating the `ReadPermission` on each field. The response will contain the subset of fields where `ReadPermission` is granted. If a field does not have an annotation, then access defaults to whatever is specified at the entity level. If the entity does not have an annotation, access defaults to whatever is specified at the package. If the package does not have an annotation, access defaults to granted.

Update

`UpdatePermission` governs whether a model can be updated by a particular user. Update is invoked when an attribute’s value is changed or values are added to or removed from a relationship. Examples of operations that will evaluate `UpdatePermission` given objects `Post` and `User` from the code snippets above:

- Changing the value of Post.published will evaluate UpdatePermission on published. Because more specific checks override less specific checks, the UpdatePermission on the entity Post will not be evaluated.
- Setting Post.author = User will evaluate UpdatePermission on Post since author does not have a more specific annotation. Because author is a bidirectional relationship, UpdatePermission will also be evaluated on the User.posts field.
- Removing Post from User.posts will trigger UpdatePermission on both the Post and User entities.
- Creating Post will *not* trigger UpdatePermission checks on any fields that are initialized in the request. However, it will trigger UpdatePermission on any bidirectional relationship fields on preexisting objects.

Create

`CreatePermission` governs whether a model can be created or a field can be initialized in a newly created model instance.Whenever a model instance is newly created, initialized fields are evaluated against `CreatePermission` rather than `UpdatePermission` .

Delete

`DeletePermission` governs whether a model can be deleted.

Share

`SharePermission` governs whether an existing model instance (one created in a prior transaction) can be assigned to another collection other than the one in which it was initially created. Basically, does a collection ‘own’ the model instance in a private sense (composition) or can it be moved or referenced by other collections (aggregation).

Graph APIs generally have two ways to reference an entity for CRUD operations. In the first mechanism, an entity is navigable through the entity relationship graph. An entity can be reached only through other entities. The alternative is to provide a mechanism to directly reference any entity by its data type and an instance identifier. The former approach is especially useful for modeling object composition (as opposed to aggregation). It enables the definition of hierarchical security. The latter approach is especially useful when directly manipulating relationships or links between edges. More specifically, to add an existing entity to a collection, it is simplest to reference this entity by its type and ID in the API request rather than defining a path to it through the entity relationship graph. Elide distinguishes between these two scenarios by tracking an object’s lineage or path through the entity relationship graph. By default, Elide explicitly denies adding an existing (not newly created) entity to a relationship with another entity if it has no lineage. For object composition, this is typically the desired behavior. For aggregation, this default can be overridden by adding an explicit SharePermission. SharePermission is always either identical to ReadPermission for the entity *or* it is explicitly disallowed.

> 原文: *http://elide.io/pages/guide/03-security.html*

Data Stores

Data Stores

`DataStores` connect a data model to a persistence layer and provide transactions that make all persistence operationsatomic in a single request.

Included Stores

Elide comes bundled with a number of data stores:

- Hibernate - Hibernate is a JPA provider that can map operations on a data model to an underlying relational database (ORM) or nosql persistence layer (OGM). Elide supports stores for Hibernate 3 and 5.
- In Memory Store - Data is persisted in a hash table on the JVM heap.
- Multiplex Store - A multiplex store delegates persistence to different underlying stores depending on the data model.
- Noop Store - A store which does nothing allowing business logic in computed attributes and life cycle hooks to entirely implement CRUD operations on the model.
  Stores can be included through the following artifact dependencies:

Hibernate Store

```
<dependency>
    <groupId>com.yahoo.elide</groupId>
    <artifactId>elide-datastore-hibernate5</artifactId>
    <version>4.0</version>
</dependency>
```

In Memory Store

```
<dependency>
    <groupId>com.yahoo.elide</groupId>
    <artifactId>elide-datastore-inmemorydb</artifactId>
    <version>4.0</version>
</dependency>
```

Multiplex Store

```
<dependency>
```

```
    <groupId>com.yahoo.elide</groupId>
    <artifactId>elide-datastore-multiplex</artifactId>
    <version>4.0</version>
</dependency>
```

Noop Store

```
<dependency>
    <groupId>com.yahoo.elide</groupId>
    <artifactId>elide-datastore-noop</artifactId>
    <version>4.0</version>
</dependency>
```

Overriding the Store

The elide-standalone artifact is the simplest way to get started with Elide. It runs an embedded Jetty container withcommon default settings including the Hibernate 5 data store.

To change the store, the `ElideStandaloneSettings` interface can be overridden to change the function which builds the `ElideSettings` object:

```
    /**
     * Elide settings to be used for bootstrapping the Elide service. By default, this method constructs an
     * ElideSettings object using the application overrides provided in this class. If this method is
overridden,
     * the returned settings object is used over any additional Elide setting overrides.
     *
     * That is to say, if you intend to override this method, expect to fully configure the ElideSettings
object to
     * your needs.
     *
     * @param injector Service locator for web service for dependency injection.
     * @return Configured ElideSettings object.
     */
    default ElideSettings getElideSettings(ServiceLocator injector) {
        DataStore dataStore = new InjectionAwareHibernateStore(
                injector, Util.getSessionFactory(getHibernate5ConfigPath(), getModelPackageName()));
        EntityDictionary dictionary = new EntityDictionary(getCheckMappings());
        return new ElideSettingsBuilder(dataStore)
                .withUseFilterExpressions(true)
                .withEntityDictionary(dictionary)
                .withJoinFilterDialect(new RSQLFilterDialect(dictionary))
                .withSubqueryFilterDialect(new RSQLFilterDialect(dictionary))
                .build();

    }
```

Custom Stores

Custom stores can be written by implenting the `DataStore` and `DataStoreTransaction` interfaces.

> 原文: *http://elide.io/pages/guide/06-datatstores.html*

Client APIs

Client APIs

Graph APIs are an evolution of web service APIs that serve and manipulate data for mobile & web applications.They have a number of characteristics that make them well suited to this task:

- Most notably, they present a **data model** as an entity relationship graph and an **accompanying schema.**
  - A well defined model allows for a consistent view of the data and a centralized way to manipulate an instance of the model or to cache it.
  - The schema provides powerful introspection capabilities that can be used to build tools to help developers understand and navigate the model.
- The API allows the client to **fetch or mutate as much or as little information in single roundtrip** between client and server. This alsoshrinks payload sizes and simplifies the process of schema evolution.

- There is a **well defined standard** for the API that fosters a community approach to development of supporting tools & best practices.
  Elide supports the two most widely adopted standards for graph APIs:

- JSON-API

- GraphQL

> 原文: *http://elide.io/pages/guide/09-clientapis.html*

Json API

Json API

JSON-API is a specification for building REST APIs for CRUD (create, read, update, and delete) operations.Similar to GraphQL:

- It allows the client to control what is returned in the response payload.
- It provided an API extension (the *patch extension* that allowed multiple mutations to the graph to occur in a single request.
  Unlike GraphQL, the JSON-API specification spells out exactly how to perform common CRUD operations including complex graph mutations.JSON-API has no standardized schema introspection. However, Elide adds this capability to any service by exporting an Open API Initiative document (formerly known as Swagger).

The json-api specification is the best reference for understanding JSON-API. The following sections describe commonly used JSON-API features as well as Elide additions for filtering, pagination, sorting, and swagger.

Hierarchical URLs

Elide generally follows the JSON-API recommendations for URL design.

There are a few caveats given that Elide allows developers control over how entities are exposed:

- Some entities may only be reached through a relationship to another entity. Not every entity is *rootable*.
- The root path segment of URLs are by default the name of the class (lowercase). This can be overridden.
- Elide allows relationships to be nested arbitrarily deep in URLs.
- Elide currently requires all individual entities to be addressed by ID within a URL. For example, consider a model with an article and a singular author which has a singular address. While unambiguous, the following is *not* allowed: /articles/1/author/address.Instead, the author must be fully qualified by ID: /articles/1/author/34/address

  ## Model Identifiers

Elide supports three mechanisms by which a newly created entity is assigned an ID:

- The ID is assigned by the client and saved in the data store.

- The client doesn’t provide an ID and the data store generates one.
- The client provides an ID which is replaced by one generated by the data store.
  When using the patch extension, the clientmust provide an ID to identify objects which are both created and added to collections in other objects. However, in some instancesthe server should have ultimate control over the ID that is assigned.
  Elide looks for the JPA `GeneratedValue` annotation to disambiguate whether or notthe data store generates an ID for a given data model. If the client also generated an ID during the object creation request, the data store ID overrides the client value.

Matching newly created objects to IDs

When using the patch extension, Elide returns object entity bodies (containing newly assigned IDs) in the order in which they were created. The client can use this order to map the object created to its server assigned ID.

Sparse Fields

JSON-API allows the client to limit the attributes and relationships that should be included in the response payloadfor any given entity. The *fields* query parameter specifies the type (data model) and list of fields that should be included.

For example, to fetch the book collection but only include the book titles:

- Request
- Response

```
/book?fields[book]=title
```

```
{
  "data": [
      {
          "attributes": {
              "title": "The Old Man and the Sea"
          },
          "id": "1",
          "type": "book"
      },
      {
          "attributes": {
              "title": "For Whom the Bell Tolls"
          },
          "id": "2",
          "type": "book"
      },
      {
          "attributes": {
              "title": "Enders Game"
          },
          "id": "3",
          "type": "book"
      }
  ]
}
```

More information about sparse fields can be found here.

Compound Documents

JSON-API allows the client to fetch a primary collection of elements but also include their relationships or their relationship’s relationships (arbitrarily nested) through compound documents. The *include* query parameter specifieswhat relationships should be expanded in the document.

The following example fetches the book collection but also includes all of the book authors. Sparse fields are usedto limit the book and author fields in the response:

- Request
- Response

```
/book?include=authors&fields[book]=title,authors&fields[author]=name
```

```
{
    "data": [
        {
            "attributes": {
                "title": "The Old Man and the Sea"
            },
            "id": "1",
            "relationships": {
                "authors": {
                    "data": [
                        {
                            "id": "1",
                            "type": "author"
                        }
                    ]
                }
            },
            "type": "book"
        },
        {
            "attributes": {
                "title": "For Whom the Bell Tolls"
            },
            "id": "2",
            "relationships": {
                "authors": {
                    "data": [
                        {
                            "id": "1",
                            "type": "author"
                        }
                    ]
                }
            },
            "type": "book"
        },
        {
            "attributes": {
                "title": "Enders Game"
            },
            "id": "3",
            "relationships": {
                "authors": {
                    "data": [
                        {
                            "id": "2",
                            "type": "author"
                        }
                    ]
                }
            },
            "type": "book"
        }
    ],
    "included": [
        {
            "attributes": {
                "name": "Ernest Hemingway"
            },
            "id": "1",
            "type": "author"
        },
        {
            "attributes": {
                "name": "Orson Scott Card"
            },
            "id": "2",
            "type": "author"
        }
    ]
}
```

More information about compound documents can be found here.

Filtering

JSON-API 1.0 is agnostic to filtering strategies. The only recommendation is that servers and clients _should_prefix filtering query parameters with the word ‘filter’.

Elide supports multiple filter dialects and the ability to add new ones to meet the needs of developers or to evolvethe platform should JSON-API standardize them. Elide’s primary dialect is RSQL

RSQL

RSQL is a query language that allows conjunction (and), disjunction (or), and parenthetic groupingof Boolean expressions. It is a superset of the FIQL language.

Because RSQL is a superset of FIQL, FIQL queries should be properly parsed.RSQL primarily adds more friendly lexer tokens to FIQL for conjunction and disjunction: ‘and’ instead of ‘;’ and ‘or’ instead of ‘,’.RSQL also adds a richer set of operators.

Filter Syntax

Filter query parameters look like `filter[TYPE]` where ‘TYPE’ is the name of the data model/entity.Any number of filter parameters can be specified provided the ‘TYPE’ is different for each parameter.

The value of any query parameter is a RSQL expression composed of predicates. Each predicate contains an attribute of the data model,an operator, and zero or more comparison values.

Filter Examples

Return all the books written by author ‘1’ with the genre exactly equal to ‘Science Fiction’:

`/author/1/book?filter[book]=genre=='Science Fiction'`

Return all the books written by author ‘1’ with the genre exactly equal to ‘Science Fiction’ *and* the title starts with ‘The’:

`/author/1/book?filter[book]=genre=='Science Fiction';title==The*`

Return all the books written by author ‘1’ with the publication date greater than a certain time *or* the genre *not* ‘Literary Fiction’or ‘Science Fiction’:

`/author/1/book?filter[book]=publishDate>1454638927411,genre=out=('Literary Fiction','Science Fiction')`

Return all the books whose title contains ‘Foo’. Include all the authors of those books whose name does not equal ‘Orson Scott Card’:

`/book?include=authors&filter[book]=title==Foo&filter[author]=name!='Orson Scott Card'`

Operators

The following RSQL operators are supported:

- =in= : Evaluates to true if the attribute exactly matches any of the values in the list.
- =out= : Evaluates to true if the attribute does not match any of the values in the list.
- ==ABC* : Similar to SQL like 'ABC%.
- ==*ABC : Similar to SQL like '%ABC.
- ==*ABC* : Similar to SQL like '%ABC%.
- =isnull=true : Evaluates to true if the attribute is null
- =isnull=false : Evaluates to true if the attribute is not null
- =lt= : Evaluates to true if the attribute is less than the value.
- =gt= : Evaluates to true if the attribute is greater than the value.
- =le= : Evaluates to true if the attribute is less than or equal to the value.
- =ge= : Evaluates to true if the attribute is greater than or equal to the value.

Values & Type Coercion

Values are specified as URL encoded strings. Elide will type coerce them into the appropriate primitive data type for the attribute filter.

Pagination

Elide supports:

- Paginating a collection by row offset and limit.
- Paginating a collection by page size and number of pages.
- Returning the total size of a collection visible to the given user.
- Returning a *meta* block in the JSON-API response body containing metadata about the collection.
- A simple way to control:
  - the availability of metadata
  - the number of records that can be paginated

Syntax

Elide allows pagination of the primary collection being returned in the response via the *page* query parameter.

The *rough* BNF syntax for the *page* query parameter is:

```
<QUERY> ::=
     "page" "[" "size" "]" "=" <INTEGER>
   | "page" "[" "number" "]" "=" <INTEGER>
   | "page" "[" "limit" "]" "=" <INTEGER>
   | "page" "[" "offset" "]" "=" <INTEGER>
   | "page" "[" "totals" "]"
```

Legal combinations of the *page* query params include:

- size
- number
- size & number
- size & number & totals
- offset
- limit
- offset & limit
- offset & limit & totals

Meta Block

Whenever a *page* query parameter is specified, Elide will return a *meta* block in theJSON-API response that contains:

- The page *number*
- The page size or *limit*
- The total number of pages (*totalPages*) in the collection
- The total number of records (*totalRecords*) in the collection.
  The values for *totalPages* and *totalRecords* are only returned if the *page[totals]* parameter was specified in the query.

Example

Paginate the book collection starting at the 4th record. Include no more than 2 books per page.Include the total size of the collection in the *meta block*:

- Request
- Response

```
/book?page[offset]=3&page[limit]=2&page[totals]
```

```
{
  "data": [
      {
          "attributes": {
              "chapterCount": 0,
              "editorName": null,
              "genre": "Science Fiction",
              "language": "English",
              "publishDate": 1464638927412,
              "title": "Enders Shadow"
          },
          "id": "4",
          "relationships": {
              "authors": {
                "data": [
                    {
                        "id": "2",
                        "type": "author"
                    }
                ]
            },
            "chapters": {
                "data": []
            },
            "publisher": {
                "data": null
            }
        },
        "type": "book"
    },
    {
        "attributes": {
            "chapterCount": 0,
            "editorName": null,
            "genre": "Science Fiction",
            "language": "English",
            "publishDate": 0,
            "title": "Foundation"
        },
        "id": "5",
        "relationships": {
            "authors": {
                "data": [
                    {
                        "id": "3",
                        "type": "author"
                    }
                ]
            },
            "chapters": {
                "data": []
            },
            "publisher": {
                "data": null
            }
        },
        "type": "book"
    }
],
"meta": {
    "page": {
        "limit": 2,
        "number": 2,
        "totalPages": 4,
        "totalRecords": 8
    }
}
```

```
}
```

Sorting

Elide supports:

- Sorting a collection by any attribute of the collection’s type.
- Sorting a collection by multiple attributes at the same time in either ascending or descending order.
- Sorting a collection by any attribute of a to-one relationship of the collection’s type. Multiple relationships can be traversed provided the path from the collection to the sorting attribute is entirely through to-one relationships.

Syntax

Elide allows sorting of the primary collection being returned in the response via the *sort* query parameter.

The *rough* BNF syntax for the *sort* query parameter is:

```
<QUERY> ::= "sort" "=" <LIST_OF_SORT_SPECS>

<LIST_OF_SORT_SPECS> = <SORT_SPEC> | <SORT_SPEC> "," <LIST_OF_SORT_SPECS>

<SORT_SPEC> ::= "+|-"? <PATH_TO_ATTRIBUTE>

<PATH_TO_ATTRIBUTE> ::= <RELATIONSHIP> <PATH_TO_ATTRIBUTE> | <ATTRIBUTE>

<RELATIONSHIP> ::= <TERM> "."

<ATTRIBUTE> ::= <TERM>
```

Sort By ID

The keyword *id* can be used to sort by whatever field a given entity uses as its identifier.

Example

Sort the collection of author 1’s books in descending order by the book’s publisher’s name:

- Request
- Response

```
/author/1/books?sort=-publisher.name
```

```
{
  "data": [
      {
          "attributes": {
              "chapterCount": 0,
              "editorName": null,
              "genre": "Literary Fiction",
              "language": "English",
              "publishDate": 0,
              "title": "For Whom the Bell Tolls"
          },
          "id": "2",
          "relationships": {
              "authors": {
                  "data": [
                      {
                          "id": "1",
                          "type": "author"
                      }
                  ]
              },
              "chapters": {
                  "data": []
              },
              "publisher": {
                  "data": {
                      "id": "2",
                      "type": "publisher"
                  }
              }
          },
          "type": "book"
      },
      {
          "attributes": {
              "chapterCount": 0,
              "editorName": null,
              "genre": "Literary Fiction",
              "language": "English",
              "publishDate": 0,
              "title": "The Old Man and the Sea"
          },
          "id": "1",
          "relationships": {
              "authors": {
                  "data": [
                      {
                          "id": "1",
                          "type": "author"
                      }
                  ]
              },
                "chapters": {
                    "data": []
                },
                "publisher": {
                    "data": {
                        "id": "1",
                        "type": "publisher"
                    }
                }
            },
            "type": "book"
        }
    ]
}
```

Bulk Writes And Complex Mutations

JSON-API supported a now-deprecated mechanism for extensions.The patch extension was a JSON-API extension that allowed muliple mutation operations (create, delete, update) to be bundled together in as single request.

Elide supports the JSON-API patch extension because it allows complex & bulk edits to the data model in the context of a single transaction.For example, the following request creates an author (earnest hemingway), multiple of his books, and his book publisher in a single request:

- Request
- Response

```
[
  {
    "op": "add",
    "path": "/author",
    "value": {
      "id": "12345678-1234-1234-1234-1234567890ab",
      "type": "author",
      "attributes": {
        "name": "Ernest Hemingway"
      },
      "relationships": {
        "books": {
          "data": [
            {
              "type": "book",
              "id": "12345678-1234-1234-1234-1234567890ac"
            },
            {
              "type": "book",
              "id": "12345678-1234-1234-1234-1234567890ad"
            }
          ]
        }
      }
    }
  },
  {
    "op": "add",
    "path": "/book",
    "value": {
      "type": "book",
      "id": "12345678-1234-1234-1234-1234567890ac",
      "attributes": {
        "title": "The Old Man and the Sea",
        "genre": "Literary Fiction",
        "language": "English"
      },
      "relationships": {
        "publisher": {
            "data": {
                "type": "publisher",
                "id": "12345678-1234-1234-1234-1234567890ae"
            }
        }
      }
    }
  },
  {
    "op": "add",
    "path": "/book",
    "value": {
      "type": "book",
      "id": "12345678-1234-1234-1234-1234567890ad",
      "attributes": {
        "title": "For Whom the Bell Tolls",
        "genre": "Literary Fiction",
        "language": "English"
      }
    }
  },
  {
    "op": "add",
    "path": "/book/12345678-1234-1234-1234-1234567890ac/publisher",
    "value": {
        "type": "publisher",
        "id": "12345678-1234-1234-1234-1234567890ae",
        "attributes": {
            "name": "Default publisher"
        }
    }
  }
]
```

```
[
  {
    "data": {
      "attributes": {
        "name": "Ernest Hemingway"
      },
      "id": "1",
      "relationships": {
        "books": {
          "data": [
            {
              "id": "1",
              "type": "book"
            },
            {
              "id": "2",
              "type": "book"
            }
          ]
        }
      },
      "type": "author"
    }
  },
  {
    "data": {
      "attributes": {
        "chapterCount": 0,
        "editorName": null,
        "genre": "Literary Fiction",
        "language": "English",
        "publishDate": 0,
        "title": "The Old Man and the Sea"
      },
      "id": "1",
      "relationships": {
        "authors": {
          "data": [
            {
              "id": "1",
              "type": "author"
            }
          ]
        },
        "chapters": {
          "data": []
        },
        "publisher": {
          "data": {
            "id": "1",
            "type": "publisher"
          }
        }
      },
      "type": "book"
    }
  },
  {
    "data": {
      "attributes": {
        "chapterCount": 0,
        "editorName": null,
        "genre": "Literary Fiction",
        "language": "English",
        "publishDate": 0,
        "title": "For Whom the Bell Tolls"
      },
      "id": "2",
      "relationships": {
        "authors": {
          "data": [
            {
              "id": "1",
              "type": "author"
            }
          ]
        },
        "chapters": {
          "data": []
        },
        "publisher": {
          "data": null
        }
      },
      "type": "book"
    }
  },
  {
    "data": {
      "attributes": {
        "name": "Default publisher"
      },
      "id": "1",
      "type": "publisher"
    }
  }
]
```

Swagger

Swagger documents can be highly customized. As a result, they are not enabled by default and instead must be initialized through code. The steps to do this are documented here.

> 原文：*http://elide.io/pages/guide/10-jsonapi.html*

GraphQL

GraphQL

GraphQL is a language specification published by Facebook for constructing graph APIs. The specification provides great flexibilityin API expression, but also little direction for best practices for common mutation operations. For example, it is silent on how to:

- Create a new object and add it to an existing collection in the same operation.
- Create a set of related, composite objects (a subgraph) and connect it to an existing, persisted graph.
- Differentiate between deleting an object vs disassociating an object from a relationship (but not deleting it).
- Change the composition of a relationship to something different.
- Reference a newly created object inside other mutation operations.
- Perform any combination of the above edits together so they can happen atomically in a single request.
  Elide offers an opinionated GraphQL API that addresses exactly how to do these things in a uniform way across your entire data model graph.

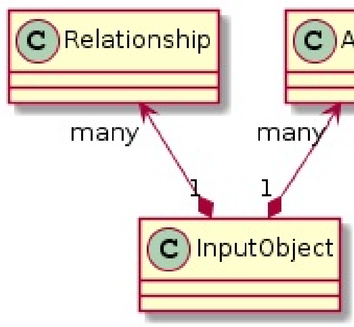
API Structure

GraphQL splits its schema into two kinds of objects:

- **Query objects** which are used to compose queries and mutations
- **Input Objects** which are used to supply input data to mutations
  The schema for both kinds of objects are derived from the entity relationship graph (defined by the JPA data model).Both contain a set of attributes and relationships. Attributes are properties of the entity.Relationships are links to other entities in the graph.

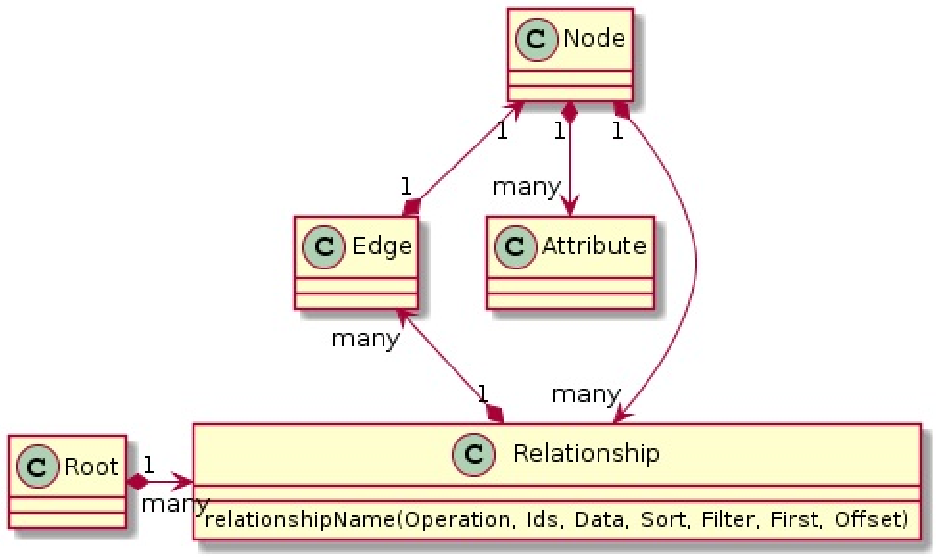
Input Objects

Input objects just contain attributes and relationship with names directly matchingthe property names in the JPA annotated model:



Query Objects

Query Objects are more complex than Input Objects since they do more than simply describe data; they mustsupport filtering, sorting, and pagination. Elide’s GraphQL structure for queries and mutations is depicted below:


Node
1
1
1
1
many
Edge
Attribute
many
1
many
Root
1
many
Relationship
relationshipName(Operation, Ids, Data, Sort, Filter, First, Offset)

Every GraphQL schema must define a root document which represents the root of the graph.In Elide, entities can be marked if they are directly navigable from the root of thegraph. Elide’s GraphQL root documents consist of *relationships* to these rootable entities.Each root relationship is named by its pluralized type name in the GraphQL root document.

All other non-rootable entities in our schema must be referenced through traversal of therelationships in the entity relationship graph.

Elide models relationships following Relay’s Connection pattern.Relationships are a collection of graph *edges*. Each edge contains a graph *node*. The *node* is an instance of adata model which in turn contains its own attributes and set of relationships.

Relationship Arguments

In GraphQL, any property in the schema can take arguments. Relationships in Elide have a standardset of arguments that either constrain the edges fetched from a relationship or supply data to a mutation:

- The **ids** parameter is a collection of node identifiers. It is used to select one or more nodes from a relationship.
- The **filter** parameter is used to build RSQL filter predicates that select zero or more nodes from a relationship.
- The **sort** parameter is used to order a relationship’s edges by one or more node attributes.
- The parameters **offset** and **first** are used to paginate a relationship across multiple API requests.
- The **op** argument describes the operation to perform on the relationship. When not provided, this argumentdefaults to a FETCH operation which simply reads the collection of edges.
- The **data** parameter is provided for operations that mutate the collection (UPSERT, UPDATE, and REPLACE), It containsa list of input objects that match the data type of the relationship. Each *data* object can be a complex subgraph which containsother objects through nested relationships.
  Entity attributes generally do not take arguments.

Relationship Operations

Elide GraphQL relationships support six operations which can be broken into two groups: data operations and id operations.The operations are separated into those that accept a *data* argument and those that accept an *ids* argument:

Operation	Data	Ids
Upsert	√	X
Update	√	X
Fetch	X	√
Replace	√	X
Remove	X	√
Delete	X	√

- The **FETCH** operation retrieves a set of objects. When a list of ids is specified, it will only extract the set of objects within therelationship with matching ids. If no ids are specified, then the entire collection of objects will be returned to the caller.
- The **DELETE** operation fully deletes an object from the system.
- The **REMOVE** operation removes a specified set (qualified by the *ids* argument) of objects from a relationship. This allows the caller to removerelationships between objects without being forced to fully delete the referenced objects.
- The **UPSERT** operation behaves much like SQL’s MERGE. Namely, if the object already exists (based on the providedid) then it will be updated. Otherwise, it will be created. In the case of updates, attributes that are not specified are left unmodified. If the *data* argument contains a complex subgraph of nested objects, nested objects will also invoke **UPSERT**.

- The **UPDATE** operation behaves much like SQL's UPDATE. Namely, if the object already exists (based on the providedid) then it will be updated. Attributes that are not specified are left unmodified. If the *data* argument contains a complex subgraph of nested objects, nested objects will also invoke **UPDATE**.
- The **REPLACE** operation is intended to replace an entire relationship with the set of objects provided in the *data* argument.**REPLACE** can be thought of as an **UPSERT** followed by an implicit **REMOVE** of everything else that was previously in the collection that the clienthas authorization to see & manipulate.

Map Data Types

GraphQL has no native support for a map data type. If a JPA data model includes a map, Elide translates this to a list of key/value pairs in the GraphQL schema.

Making Calls

All calls must be HTTP `POST` requests made to the root endpoint. This specific endpoint will depend on where you mount the provided servlet.For example, if the servlet is mounted at `/graphql` , all requests should be sent as:

```
POST https://yourdomain.com/graphql
```

Example Data Model

All subsequent query examples are based on the following data model including `Book` , `Author` , and `Publisher` :

- Book.java
- Author.java
- Publisher.java

```
@Entity
@Table(name = "book")
@Include(rootLevel = true)
public class Book {
    @Id public long id;
    public String title;
    public String genre;
    public String language;
    @ManyToMany
    public Set<Author> authors;
    @ManyToOne
    Publisher publisher;
}
```

```java
@Entity
@Table(name = "author")
@Include(rootLevel = false)
public class Author {
    @Id public long id;
    public String name;
    @ManyToMany
    public Set<Book> books;
}
```

```java
@Entity
@Table(name = "publisher")
@Include(rootLevel = false)
public class Publisher {
    @Id public long id;
    public String name;
    @OneToMany
    public Set<Book> books;
}
```

Filtering

Elide supports filtering relationships for any *FETCH* operation by passing a RSQL expression in the *filter* parameter for the relationship. RSQL is a query language that allows conjunction (and), disjunction (or), and parenthetic groupingof boolean expressions. It is a superset of the FIQL language.

RSQL predicates can filter attributes in:

- The relationship model
- Another model joined to the relationship model through to-one relationships
  To join across relationships, the attribute name is prefixed by one or more relationship names separated by period ('.')

Operators

The following RSQL operators are supported:

- =in= : Evaluates to true if the attribute exactly matches any of the values in the list.
- =out= : Evaluates to true if the attribute does not match any of the values in the list.
- ==ABC* : Similar to SQL like 'ABC%.
- ==*ABC : Similar to SQL like '%ABC.
- ==*ABC* : Similar to SQL like '%ABC%.
- =isnull=true : Evaluates to true if the attribute is null

- =isnull=false : Evaluates to true if the attribute is not null
- =lt= : Evaluates to true if the attribute is less than the value.
- =gt= : Evaluates to true if the attribute is greater than the value.
- =le= : Evaluates to true if the attribute is less than or equal to the value.
- =ge= : Evaluates to true if the attribute is greater than or equal to the value.

Examples

- Filter books by title equal to ‘abc’ *and* genre starting with ‘Science’:"title=='abc';genre=='Science*'
- Filter books with a publication date greater than a certain time *or* the genre is *not* ‘Literary Fiction’or ‘Science Fiction’:publishDate>1454638927411,genre=out=('Literary Fiction','Science Fiction')
- Filter books by the publisher name contains XYZ:publisher.name==*XYZ*

Pagination

Any relationship can be paginated by providing one or both of the following parameters:

- **first** - The number of items to return per page.
- **offset** - The number of items to skip.

Relationship Metadata

Every relationship includes information about the collection (in addition to a list of edges) that can be requested on demand:

- **endCursor** - The last record offset in the current page (exclusive).
- **startCursor** - The first record offset in the current page (inclusive).
- **hasNextPage** - Whether or not more pages of data exist.
- **totalRecords** - The total number of records in this relationship across all pages. These properties are contained within the the *pageInfo* structure:

```
{
  pageInfo {
    endCursor
    startCursor
    hasNextPage
    totalRecords
  }
}
```

Sorting

Any relationship can be sorted by attributes in:

- The relationship model
- Another model joined to the relationship model through to-one relationships To join across relationships, the attribute name is prefixed by one or more relationship names separated by period (‘.’)

It is also possible to sort in either ascending or descending order by prependingthe attribute expression with a ‘+’ or ‘-‘ character. If no order character is provided, sort order defaults to ascending.

A relationship can be sorted by multiple attributes by separating the attribute expressions by commas: ‘,’.

Model Identifiers

Elide supports three mechanisms by which a newly created entity is assigned an ID:

- The ID is assigned by the client and saved in the data store.
- The client doesn’t provide an ID and the data store generates one.
- The client provides an ID which is replaced by one generated by the data store. When using *UPSERT*, the clientmust provide an ID to identify objects which are both created and added to collections in other objects. However, in some instancesthe server should have ultimate control over the ID that is assigned. Elide looks for the JPA `GeneratedValue` annotation to disambiguate whether or notthe data store generates an ID for a given data model. If the client also generated an ID during the object creation request, the data store ID overrides the client value.

Matching newly created objects to IDs

When using *UPSERT*, Elide returns object entity bodies (containing newly assigned IDs) in the order in which they were created - assuming all the entities were newly created (and not mixedwith entity updates in the request). The client can use this order to map the object created to its serverassigned ID.

FETCH Examples

Fetch All Books

Include the id, title, genre, & language in the result.

- Request

- Response

```
{
  book {
    edges {
      node {
        id
        title
        genre
        language
      }
    }
  }
}
```

```
{
  "book": {
    "edges": [
      {
        "node": {
          "id": "1",
          "title": "Libro Uno",
          "genre": null,
          "language": null
        }
      },
      {
        "node": {
          "id": "2",
          "title": "Libro Dos",
          "genre": null,
          "language": null
        }
      },
      {
        "node": {
          "id": "3",
          "title": "Doctor Zhivago",
          "genre": null,
          "language": null
        }
      }
    ]
  }
}
```

Fetch Single Book

Fetches book 1. The response includes the id, title, and authors.For each author, the response includes its id & name.

- Request
- Response

```
{
  book(ids: ["1"]) {
    edges {
      node {
        id
        title
        authors {
          edges {
            node {
              id
              name
            }
          }
        }
      }
    }
  }
}
```

```
{
  "book": {
    "edges": [
      {
        "node": {
          "id": "1",
          "title": "Libro Uno",
          "authors": {
            "edges": [
              {
                "node": {
                  "id": "1",
                  "name": "Mark Twain"
                }
              }
            ]
          }
        }
      }
    ]
  }
}
```

Filter All Books

Fetches the set of books that start with ‘Libro U’.

- Request

- Response

```
{
  book(filter: "title==\"Libro U*\"") {
    edges {
      node {
        id
        title
      }
    }
  }
}
```

```
{
  "book": {
    "edges": [
      {
        "node": {
          "id": "1",
          "title": "Libro Uno"
        }
      }
    ]
  }
}
```

Paginate All Books

Fetches a single page of books (1 book per page), starting at the 2nd page.Also requests the relationship metadata.

- Request
- Response

```
{
  book(first: "1", after: "1") {
    edges {
      node {
    id
    title
      }
    }
    pageInfo {
      totalRecords
      startCursor
      endCursor
      hasNextPage
    }
  }
}
```

```
{
  "book": {
    "edges": [
      {
    "node": {
      "id": "2",
      "title": "Libro Dos"
    }
      }
    ],
    "pageInfo": {
      "totalRecords": 3,
      "startCursor": "1",
      "endCursor": "2",
      "hasNextPage": true
    }
  }
}
```

Sort All Books

Sorts the collection of books first by their publisher id (descending) and then by the book id (ascending).

- Request
- Response

```
{
  book(sort: "-publisher.id,id") {
    edges {
      node {
        id
        title
        publisher {
          edges {
            node {
              id
            }
          }
        }
      }
    }
  }
}
```

```
{
  "book": {
    "edges": [
      {
```

```
  "node": {
   "id": "3",
   "title": "Doctor Zhivago",
   "publisher": {
    "edges": [
     {
    "node": {
     "id": "2"
    }
     }
    ]
   }
  }
   },
   {
  "node": {
   "id": "1",
   "title": "Libro Uno",
   "publisher": {
    "edges": [
     {
    "node": {
     "id": "1"
    }
     }
    ]
   }
  }
   },
   {
  "node": {
   "id": "2",
   "title": "Libro Dos",
   "publisher": {
    "edges": [
     {
    "node": {
     "id": "1"
    }
     }
    ]
   }
  }
   }
  ]
 }
}
```

Schema Introspection

Fetches the entire list of data types in the GraphQL schema.

- Request
- Response

```
{
  __schema {
    types {
      name
    }
  }
}
```

```
{
  "__schema": {
    "types": [
      {
        "name": "root"
      },
      {
        "name": "noshare"
      },
      {
        "name": "__edges__noshare"
      },
      {
        "name": "__node__noshare"
      },
      {
        "name": "id"
      },
      {
        "name": "__pageInfoObject"
      },
      {
        "name": "Boolean"
      },
      {
        "name": "String"
      },
      {
        "name": "Long"
      },
      {
        "name": "com.yahoo.elide.graphql.RelationshipOp"
      },
      {
        "name": "noshareInput"
      },
      {
        "name": "ID"
      },
      {
        "name": "book"
      },
      {
        "name": "__edges__book"
      },
      {
        "name": "__node__book"
      },
      {
        "name": "authorInput"
      },
      {
        "name": "example.AddressInputInput"
      },
      {
        "name": "example.Author$AuthorType"
      },
      {
        "name": "bookInput"
      },
      {
        "name": "publisherInput"
      },
      {
        "name": "pseudonymInput"
      },
      {
        "name": "author"
      },
      {
        "name": "__edges__author"
      },
      {
        "name": "__node__author"
      },
      {
        "name": "example.Address"
      },
      {
        "name": "publisher"
      },
      {
        "name": "__edges__publisher"
      },
      {
        "name": "__node__publisher"
      },
      {
        "name": "pseudonym"
      },
      {
        "name": "__edges__pseudonym"
      },
      {
          "name": "__node__pseudonym"
        },
        {
          "name": "__Schema"
        },
        {
          "name": "__Type"
        },
        {
          "name": "__TypeKind"
        },
        {
          "name": "__Field"
        },
        {
          "name": "__InputValue"
        },
        {
          "name": "__EnumValue"
        },
        {
          "name": "__Directive"
        },
        {
          "name": "__DirectiveLocation"
        }
      ]
    }
}
```

UPSERT Examples

Create and add new book to an author

Creates a new book and adds it to Author 1.The author’s id and list of newly created books is returned in the response. For each newly created book, only the title is returned.

- Request
- Response

```
mutation {
  author(ids: ["1"]) {
    edges {
      node {
        id
        books(op: UPSERT, data: {title: "Book Numero Dos"}) {
        edges {
            node {
```

```
            title
          }
        }
      }
    }
  }
}
}
```

```
{
  "author": {
    "edges": [
      {
        "node": {
          "id": "1",
          "books": {
            "edges": [
              {
                "node": {
                  "title": "Book Numero Dos"
                }
              }
            ]
          }
        }
      }
    ]
  }
}
```

Update the title of an existing book

Updates the title of book 1 belonging to author 1.The author’s id and list of updated books is returned in the response. For each updated book, only the title is returned.

- Request
- Response

```
mutation {
  author(ids: ["1"]) {
    edges {
      node {
        id
        books(op:UPSERT, data: {id: "1", title: "abc"}) {
          edges {
            node {
              id
              title
            }
          }
        }
      }
    }
  }
}
```

```
{
  "author": {
    "edges": [
      {
        "node": {
          "id": "1",
          "books": {
            "edges": [
              {
                "node": {
                  "id": "1",
                  "title": "abc"
                }
              }
            ]
          }
        }
      }
    ]
  }
}
```

UPDATE Examples

Updates author 1’s name and simultaneously updates the titles of books 2 and 3.

- Request
- Response

```
mutation {
  author(op:UPDATE, data: {id: "1", name: "John Snow", books: [{id: "3", title: "updated again"}, {id: "2", title: "newish title"}]}) {
    edges {
      node {
    id
    name
    books(ids: ["3"]) {
      edges {
        node {
          title
        }
      }
    }
      }
    }
  }
}
```

```
{
  "author": {
    "edges": [
      {
    "node": {
      "id": "1",
      "name": "John Snow",
      "books": {
        "edges": [
          {
        "node": {
          "title": "updated again"
        }
          }
        ]
      }
    }
      }
    ]
  }
}
```

DELETE Examples

Deletes books 1 and 2. The id and title of the remaining books are returned in the response.

- Request
- Response

```
mutation {
  book(op:DELETE, ids: ["1", "2"]) {
    edges {
      node {
        id
        title
      }
    }
  }
}
```

```
{
  "book": {
    "edges": [
    ]
  }
}
```

REMOVE Example

Removes books 1 and 2 from author 1. Author 1 is returned with the remaining books.

- Request
- Response

```
mutation {
  author(ids: ["1"]) {
    edges {
      node {
        books(op:REMOVE, ids: ["1", "2"]) {
          edges {
            node {
              id
              title
            }
          }
        }
      }
    }
  }
}
```

```
{
  "author": {
    "edges": [
      {
    "node": {
      "books": {
        "edges": [
        ]
      }
    }
      }
    ]
  }
}
```

REPLACE Example

Replaces the set of authors for *every* book with the set consisting of:

- An existing author (author 1)
- A new author
  The response includes the complete set of books (id & title) and their new authors (id & name).
- Request
- Response

```
mutation {
  book {
    edges {
      node {
    id
    title
    authors(op: REPLACE, data:[{name:"My New Author"},{id:"1"}]) {
      edges {
        node {
          id
          name
        }
      }
    }
      }
    }
  }
}
```

```
{
  "book": {
    "edges": [
      {
    "node": {
      "id": "1",
      "title": "Libro Uno",
      "authors": {
        "edges": [
          {
        "node": {
          "id": "3",
          "name": "My New Author"
        }
          },
          {
        "node": {
          "id": "1",
          "name": "Mark Twain"
        }
          }
        ]
      }
    }
      },
      {
    "node": {
      "id": "2",
      "title": "Libro Dos",
      "authors": {
        "edges": [
          {
        "node": {
          "id": "4",
          "name": "My New Author"
        }
          },
          {
        "node": {
          "id": "1",
          "name": "Mark Twain"
        }
          }
        ]
      }
    }
      },
      {
    "node": {
      "id": "3",
      "title": "Doctor Zhivago",
      "authors": {
        "edges": [
          {
        "node": {
          "id": "5",
          "name": "My New Author"
        }
          },
          {
        "node": {
          "id": "1",
          "name": "Mark Twain"
        }
          }
        ]
      }
    }
      }
    ]
  }
}
```

> 原文: *http://elide.io/pages/guide/11-graphql.html*

Audit

Audit

Audit assigns semantic meaning to CRUD operations for the purposes of logging and audit. For example, we may want to log when users change their password or when an account is locked. Both actions are mutations on a user entity that update different fields. Audit can assign these actions to parameterized, human readable logging statements that can be logged to a file, written to a database, etc.

Core Concepts

A model’s **lineage** is the path taken through the entity relationship graph to reach it.A model and every prior model in its lineage are fully accessible to parameterize audit logging in Elide.

Annotations

Elide audits operations on classes and class fields marked with the `Audit` annotation.

The `Audit` annotation takes several arguments:

- The CRUD action performed (CREATE, DELETE, or UPDATE).
- An operation code which uniquely identifies the semantic meaning of the action.
- The statement to be logged. This is a template string that allows ‘{}’ variable substitution.
- An ordered list of Unified Expression Language expressions that are used to subtitute ‘{}’ in the log statement. Elide binds the model that is being audited and every model in its lineage to variables that are accessible to the UEL expressions. The variable names map to model’s type (typically the class name).

Example

Let’s say I have a simple *user* entity with a *password* field. I want to audit whenever the password is changed. The user is accessed via the URL path ‘/company/53/user/21’. I could annotate this action as follows:

```
@Entity
@Include
public class User {
    @Audit(action = Audit.Action.UPDATE,
           operation = 572,
           logStatement = "User {0} from company {1} changed password.",
```

```java
            logExpressions = {"${user.userid}", "${company.name}"})
    private String password;
    private String userid;
}
```

Elide binds the `User` object to the variable name *user* and the `Company` object to the variable name *company*. The `Company` object is bound because it belongs to the `User` object’s lineage.

Customizing Logging

Customizing audit functionality in elide requires two steps:

- Define audit annotations on JPA entity classes and fields.
- Provide a Logger implementation to customize the handling of audit triggers. The default logger simply logs to slf4j.

Logger Implementation

A customized logger extends the following abstract class:

```java
public abstract class AuditLogger {
    public void log(LogMessage message);
    public abstract void commit(RequestScope requestScope) throws IOException;
}
```

> 原文: *http://elide.io/pages/guide/12-audit.html*

Swagger

Swagger

Overview

Elide supports the generation of Swagger documentation from Elide annotated beans. Specifically, it generates a JSON documentconforming to the swagger specification that can be used by tools like Swagger UI (among others) to explore, understand, and compose queries againstyour Elide API.

Features Supported

- **JaxRS Endpoint** - Elide ships with a customizable JaxRS endpoint that can publish one or more swagger documents.
- **Path Discovery** - Given a set of entities to explore, Elide will generate the minimum, cycle-free, de-duplicated set of URL paths in the swagger document.
- **Filter by Primitive Attributes** - All *GET* requests on entity collections include filter parameters for each primitive attribute.
- **Prune Fields** - All *GET* requests support JSON-API sparse fields query parameter.
- **Include Top Level Relationships** - All *GET* requests support the ability to include direct relationships.
- **Sort by Attribute** - All *GET* requests support sort query parameters.
- **Pagination** - All *GET* requests support pagination query parameters.
- **Permission Exposition** - Elide permissions are exported as documentation for entity schemas.

  ## Getting Started

Maven

Pull in the following elide dependencies :

```xml
<dependency>
  <groupId>com.yahoo.elide</groupId>
  <artifactId>elide-swagger</artifactId>
</dependency>

<dependency>
  <groupId>com.yahoo.elide</groupId>
  <artifactId>elide-core</artifactId>
</dependency>
```

```
```

Pull in swagger core :

```
<dependency>
  <groupId>io.swagger</groupId>
  <artifactId>swagger-core</artifactId>
</dependency>
```

Basic Setup

Create and initialize an entity dictionary.

```
EntityDictionary dictionary = new EntityDictionary(Maps.newHashMap());

dictionary.bindEntity(Book.class);
dictionary.bindEntity(Author.class);
dictionary.bindEntity(Publisher.class);
```

Create a swagger info object.

```
Info info = new Info().title("My Service").version("1.0");
```

Initialize a swagger builder.

```
SwaggerBuilder builder = new SwaggerBuilder(dictionary, info);
```

Build the document & convert to JSON.

```
Swagger document = builder.build();
String jsonOutput = SwaggerBuilder.getDocument(document);
```

Supporting OAuth

If you want swagger UI to acquire & use a bearer token from an OAuth identity provider, you can configurethe swagger document similar to:

```
SecuritySchemeDefinition oauthDef = new OAuth2Definition().implicit(CONFIG_DATA.zuulAuthorizeUri());
SecurityRequirement oauthReq = new SecurityRequirement().requirement("myOuath");

SwaggerBuilder builder = new SwaggerBuilder(entityDictionary, info);
Swagger document = builder.build();
    .basePath("/my/url/path")
    .securityDefinition("myOauth", oauthDef)
    .security(oauthReq)
    .scheme(Scheme.HTTPS));
```

Adding a global parameter

A query or header parameter can be added globally to all Elide API endpoints:

```
HeaderParameter oauthParam = new HeaderParameter()
    .name("Authorization")
    .type("string")
    .description("OAuth bearer token")
    .required(false);

SwaggerBuilder crashBuilder = new SwaggerBuilder(dictionary, info)
    .withGlobalParameter(oauthParam);
```

Adding a global response code

An HTTP response can be added globally to all Elide API endpoints:

```
Response rateLimitedResponse = new Response().description("Too Many Requests");

SwaggerBuilder crashBuilder = new SwaggerBuilder(dictionary, info)
    .withGlobalResponse(429, rateLimitedResponse);
```

Performance

Path Generation

The Swagger UI is very slow when the number of generated URL paths exceeds a few dozen. For large, complex data models, it is recommended to generate separate swagger documents for subgraphs of the model.

```
Set<Class<?>> entities = Sets.newHashSet(
    Book.class,
    Author.class,
    Publisher.class
);

SwaggerBuilder coreBuilder = new SwaggerBuilder(dictionary, info)
    .withExplicitClassList(entities);
```

In the above example, swagger will only generate paths that exclusively traverse the provided set of entities.

Document Size

The size of the swagger document can be reduced significantly by limiting the number of filter operators that are used to generate query parameterdocumentation.

```
SwaggerBuilder crashBuilder = new SwaggerBuilder(dictionary, info)
   .withFilterOps(Sets.newHashSet(Operator.IN));
```

In the above example, filter query parameters are only generated for the *IN* operator.

> 原文: *http://elide.io/pages/guide/13-swagger.html*

Test

Test

The surface of an Elide web service can become quite large given the numerous ways of navigating a complex data model. This can make testing daunting.Elide has a sibling project for testing the authorization logic of your service.

Overview

The test framework allows the developer to redefine the authorization rules for different classes of users in a domain specific language.

Provided some test data for a `DataStore` , the framework builds a graph (cycles removed) of every possibleway to navigate the test data originating from the rootable entities.

For every collection and entity in the graph, it tests all combinations of CRUD operations - comparing the result returned from Elide with the expected resultdefined in the domain specific language. Any mismatches are reported as test failures.

Domain Specific Language

The DSL is structured as a gherkin feature file with the following elements:

Exposed Entities

Exposed entities is a table describing which JPA entities are exposed through Elide and which of those entities are rootable.

```
Given exposed entities
  | EntityName     | Rootable |
  | parent         | true     |
  | child          | false    |
```

Users

There will be different classes of users who have access to the system. A class of users are the set of users who share the same authorizationpermissions. Ideally, the developer should define a `user` for each permutation of authorization permissions they want to test.

```
And users
  # Amalberti father
```

```
        | Emmanuel |
        # Bonham parents
        | Mo       |
        | Margery  |
```

The concept of users is opaque to the security test framework. It has no concept of authentication or user identity. Instead, the developer providesa concrete implementation of a `UserFactory` which constructs the user objects your authorization code expects:

```
public interface UserFactory {
        User makeUser(String alias);
}
```

Associated Permissions

Associated permissions is a table that defines which entities a given user class has access to. For each set of entities, it alsodefines which CRUD operations are allowed.

```
    And associated permissions
        | UserName  | EntityName | ValidIdsList             | EntityPermissions         | RestrictedReadFields   | RestrictedWriteFields |
         ########### ############ ##############            ########################### ####################### #######################
        | Mo        | parent     | Mo                       | Create,Read,Update        |                        | deceased              |
        | Mo        | parent     | Mo's Spouse              | Create,Read,Update        | otherSpouses           | deceased              |
        | Mo        | parent     | Emmanuel,Goran,Hina      | Read,Update               | otherSpouses           | [EXCLUDING] friends |
```

UserName

This column identifies the user class.

EntityName

This column identifies the JPA entity. The name must match the name exposed via Elide.

ValidIdsList

There is a grammar which defines the syntax for this column. In short, it can be one of the following:

- A comma separated list of entity IDs. The IDs much match the test data in the DataStore.
- The keyword [ALL] which signifies all the IDs found in the test data for the given entity.
- An expression [type.collection] where type is another entity and collection is a

relationship inside of type that contains elements of type EntityName. This expression semantically means: ‘If the user has any access to an entity of type type, they should have access to everything inside that entity’s collection collection.’ More concretely, this expression will get expanded to a comma separated list of IDs by first expanding type to the set of entities of type type the user class has access to. For each of these entities, it will then expand to the list of IDs contained within collection. Expressions of this type can be combined within lists of comma separated IDs.
Aliases can also be defined to represent one or more comma separated IDs, These aliases are expanded in the grammar expression prior to parsing. Aliases are defined in the gherkin file:

```
And aliases
  | Mo                   | 1 |
  | Margery              | 2 |
  | Mo's Spouse          | Margery |
  | Margery's Spouse     | Mo |
  | Margery's Ex         | Emmanuel |
  | Emmanuel             | 3 |
  | Emmanuel's Ex        | Margery |
  | Goran                | 4 |
  | Bonham Children      | 1,2,3 |
  | Amalberti Children   | 4,5,6 |
  | Tang Children        | 7 |
```

Aliases can reference other aliases.

EntityPermissions

Entity Permissions are the list of CRUD permissions that are allowed for the given list of entities for the given user. They should be definedas a comma separated list of the keywords ‘Create’, ‘Read’, ‘Update’, and ‘Delete’.

RestrictedReadFields & RestrictedWriteFields

The list of entity fields the given user class should not be able to read or write respectively.There is a grammar which defines the syntax for this column. In short, it can be one of the following:

- A comma separated list of field names.
- The keyword [ALL] which signifies all fields.
- The keyword [EXCLUDING] followed by a list of fields. This inverts the column to a white list of allowed fields instead of a blacklist of excluded fields.

Disabled Test Ids

```
And disabled test ids
  | CreateRelation:child#5/relationships/playmates:Denied=[1,2,3] |
      | CreateRelation:child#6/relationships/playmates:Denied=[3]     |

```

If you want to disable a test from running, you can do so by adding ‘name’ of the test to this list. The ‘name’ of the tests are displayedin the output of test framework executions.

Complete Example

A full example of the configuration DSL can be found here

Using The Framework

Using the framework can be broken down into the following steps:

- Create a feature file using the described DSL.
- Write a Java program that does the following:
  - Create a DataStore
  - Create test data.
  - Create a UserFactory.
  - Create a ValidationDriver and invoke execute.

```java
/* The data store we use is orthogonal to the correctness of security checks */
DataStore dataStore = new InMemoryDB();

/* Construct test data and persist to the store */
initializeTestData(dataStore);

String featureFile = "SampleConfig.feature";

/* Pass in a NOOP logger */
AuditLogger logger = new AuditLogger() {
    @Override
    public void commit() throws IOException {}
};


/* This class will instantiate our users */
UserFactory userFactory  = new MyUserFactory(dataStore);

/* Create and execute the test driver */
ValidationDriver driver = new ValidationDriver(featureFile, userFactory, dataStore, logger);

/* Results per user can be printed or processed in some other way */
Map<UserProfile, List<ValidationResult>> results = driver.execute();
```

Given that the goal of the test framework is limited to testing authorization code (security check logic), the `DataStore` used to furnish test datashould be orthogonal to the correctness of the tests.

Fail On Missing Tests

The `ValidationDriver` takes an optional boolean parameter (`failOnMissingTests`) that will fail the test execution if there is no test datafor any entity exposed via Elide. If this parameter is not set, it will simply log the absence of test data.

> 原文: *http://elide.io/pages/guide/14-test.html*

Annotation Overview

Annotation Overview

Elide exposes data models using a set of annotations. To describe relational modeling, we rely on the well-adopted JPA annotations. For exposition and security, we rely on custom Elide annotations. A comprehensive list of supported Elide annotations is below.

- Audit
- Audits
- ComputedAttribute
- ComputedRelationship
- CreatePermission
- DeletePermission
- Exclude
- Include
- OnCreatePostCommit
- OnCreatePreCommit
- OnCreatePreSecurity
- OnDeletePostCommit
- OnDeletePreCommit
- OnDeletePreSecurity
- OnReadPostCommit
- OnReadPreCommit
- OnReadPreSecurity
- OnUpdatePostCommit
- OnUpdatePreCommit
- OnUpdatePreSecurity
- Paginate
- ReadPermission
- SharePermission
- UpdatePermission

@Audit" class="reference-link">@Audit

- Description

Enables audit logging for a particular package, class, method, or field whenever a specified action takes place via Elide. It takes advantage of Elide's Audit logging capabilities.

- Application Level

- Class
- Field
- Method
- Package

- Parameters

- action
  - **Description:** The set of performed action(s) upon which audit should be triggered.
  - **Type:**Action[]
  - **Required:** false
  - **Default Value:** {Action.CREATE, Action.UPDATE, Action.DELETE}
- logExpressions
  - **Description:** Unified expression language expressions that will be evaluated and substituted into the logging template.
  - **Type:**String[]
  - **Required:** false
  - **Default Value:** ""
- logStatement
  - **Description:** Logging string template passed to audit logger for fired audit event.
  - **Type:**String
  - **Required:** false
  - **Default Value:** ""
- operation
  - **Description:** Operation code to pass to audit logger for fired audit event.
  - **Type:**Integer
  - **Required:** false
  - **Default Value:** -1

@Audits

- Description

Enables a set of audit logging annotations to be applied to a particular package, class, method, or field whenever a specified action takes place through Elide.

- Application Level

- Class
- Field
- Method
- Package

- Parameters

- value

  - **Description:** A set of @Audit annotations.
  - **Type:**Audit[]
  - **Required:** true
  - **Default Value:***None*

@ComputedAttribute

- Description

Marks a method or field as a computed attribute that should be exposed via Elide regardless of whether or not it is marked as Transient.

- Application Level

- Field
- Method

- Parameters

- *None*

@ComputedRelationship

- Description

Marks a method or field as a computed relationship that should be exposed via Elide regardless of whether or not it is marked as Transient.

- Application Level

- Field
- Method

- Parameters

- *None*

@CreatePermission

- Description

Define security rules for creating an object through Elide. See the security section for more information.

- Application Level

- Class
- Field
- Method
- Package

- Parameters

- expression
    - **Description:** A security expression parsed by Elide security. See the security section for more information.
    - **Type:**String
    - **Required:** true
    - **Default Value:***None*

@DeletePermission

- Description

Define security rules for deleting an object through Elide. See the security section for more information.

- Application Level

- Class
- Field
- Method
- Package

- Parameters

- expression
    - **Description:** A security expression parsed by Elide security. See the security section for more information.
    - **Type:**String
    - **Required:** true
    - **Default Value:***None*

@Exclude

- Description

Marks that a given field or entity should not be exposed through Elide.

- Application Level

- Class
- Field
- Method
- Package

- Parameters

- *None*

@Include

- Description

Marks that a given entity should be exposed through Elide.

- Application Level

- Class
- Package

- Parameters

- rootLevel
  - **Description:** Whether or not the entity is accessible as a "rootable" entity. Namely, if this collection of objects can be queried directly or whether or not it must be queried through a relationship.
  - **Type:**Boolean
  - **Required:** false
  - **Default Value:** false
- type
  - **Description:** The API-exposed name for a particular entity type.
  - **Type:**String
  - **Required:** false
  - **Default Value:***Camel-cased name of the entity class*

@OnCreatePostCommit

- Description

A lifecycle hook that executes on a create action after the transaction has committed successfully.

- Application Level

- Method

- Parameters

- *None*

@OnCreatePreCommit

- Description

A lifecycle hook that executes on a create action after security has run, but before the transaction has committed.

- Application Level

- Method
- Parameters
- *None*

@OnCreatePreSecurity

- Description

A lifecycle hook that executes on a create action before the security checks have run.

- Application Level
- Method
- Parameters
- *None*

@OnDeletePostCommit

- Description

A lifecycle hook that executes on a delete action after the transaction has committed successfully.

- Application Level
- Method
- Parameters
- *None*

@OnDeletePreCommit

- Description

A lifecycle hook that executes on a delete action after security has run, but before the transaction has committed.

- Application Level
- Method
- Parameters
- *None*

@OnDeletePreSecurity

- Description

A lifecycle hook that executes on a delete action before the security checks have run.

- Application Level

- Method

- Parameters

- *None*

@OnReadPostCommit

- Description

A lifecycle hook that executes on a read action after the transaction has committed successfully.

- Application Level

- Method

- Parameters

- *None*

@OnReadPreCommit

- Description

A lifecycle hook that executes on a read action after security has run, but before the transaction has committed.

- Application Level

- Method

- Parameters

- *None*

@OnReadPreSecurity

- Description

A lifecycle hook that executes on a read action before the security checks have run.

- Application Level

- Method

- Parameters
- *None*

@OnUpdatePostCommit

- Description

A lifecycle hook that executes on a update action after the transaction has committed successfully.

- Application Level
- Method
- Parameters
- *None*

@OnUpdatePreCommit

- Description

A lifecycle hook that executes on a update action after security has run, but before the transaction has committed.

- Application Level
- Method
- Parameters
- *None*

@OnUpdatePreSecurity

- Description

A lifecycle hook that executes on a update action before the security checks have run.

- Application Level
- Method
- Parameters
- *None*

@Paginate

- Description

Apply specific pagination rules to a specific entity.

- Application Level

- Class

- Parameters

- countable
  - **Description:** Whether or not the API will respond to request for page totals of this entity type.
  - **Type:**Boolean
  - **Required:** false
  - **Default Value:** true
- defaultLimit
  - **Description:** If the client does not specify a page size, this is the number of elements that will be returned.
  - **Type:**Integer
  - **Required:** false
  - **Default Value:** 500
- maxLimit
  - **Description:** The maximum number of records a user can request at a given time for a particular entity.
  - **Type:**Integer
  - **Required:** false
  - **Default Value:** 10000

@ReadPermission

- Description

Define security rules for reading an object through Elide. See the security section for more information.

- Application Level

- Class
- Field
- Method
- Package

- Parameters

- expression
  - **Description:** A security expression parsed by Elide security. See the security section for more information.
  - **Type:**String
  - **Required:** true
  - **Default Value:***None*

@SharePermission

- Description

Enable sharing permissions on an object through Elide. See the security section for more information.

- Application Level

- Class
- Method
- Package

- Parameters

- shareable
  - **Description:** Enable or disable shareability for an object. Shareable permissions are evaluated identically to an entity's read permissions. See the security section for more information.
  - **Type:**Boolean
  - **Required:** false
  - **Default Value:** true

@UpdatePermission

- Description

Define security rules for updating an object through Elide. See the security section for more information.

- Application Level

- Class
- Field
- Method
- Package

- Parameters

- expression
  - **Description:** A security expression parsed by Elide security. See the security section for more information.
  - **Type:**String
  - **Required:** true
  - **Default Value:***None*

> 原文: *http://elide.io/pages/guide/15-annotations.html*