Katharsis JSON-API Documentation Documentation

Release 3.0.0

Ioan Eugen Stan <ieugen@netdava.com>, Patryk Orwat, Błażej Kr

Apr 05, 2017

User Documentation

Katharsis provides means to easily expose resources over the REST interface. If you want to take a deeper look on the concepts behind Katharsis, this is page for you!

For the latest documentation please visit http://katharsis-jsonapi.readthedocs.io/en/latest/ .

The main documentation for the site is organized into a couple sections:

- *User Documentation*
- *Contributor documentation*

Information about how to contribute is also available:

- *Contributor documentation*

CHAPTER 1

How to use Katharsis in your applications

Katharsis provides means to easily expose resources over the REST interface. If you want to take a deeper look on the concepts behind Katharsis, this is page for you!

Release Notes

3.0

- a new, more powerful JsonApiRelation annotation easing the declarations of relationships.
- QueryParams and Repository annotations have been deprecated (but resource annotations not!) and moved to a legacy package. Interface-based repositories and QuerySpec are the recommended APIs.
- major cleanup release with more test cases.
- some few package changes to non-internal classes and interfaces.
- serialization layer has been completely rewritten and is more robust/easier to understand.
- a new DocumentFilter has been introduced that allows to modify incoming requests and responses.
- many classes considered to be non-public have been moved to internal packages.
- a rewritten servlet integratino for Katharsis based on KatharsisBoot.

Features deprecated with this 3.0 releases remain support for quite some time to come. As such *Katharsis 3.0 remains backwards compatible with 2.x*. Some small migrations steps are necessary:

- fix package imports
- Replace all usages of QuerySpec<Xy>Repository with <Xy>RepositoryV2.

In the medium/long-term make sure to avoid and drop the usage of any deprecated legacy API (contained in the io.katharsis.legacy packages):

- make use of interface instead of annotation-based repositories.

- make use of QuerySpec instead of QueryParams. You can perform this step gradually. Once that is completed, you can also to the next step.
- move away from DefaultQueryParamsParser in favor of DefaultQuerySpecDeserializer. Note that this step can only be done once all the QueryParams-based repositories have been migrated to QuerySpec. This will make your repositories fully JSON-API compliant (`sort` parameter now adheres to the official specification, see below for examples).
- (recommendation) Avoid the use of `KatharsisProperties.RESOURCE_SEARCH_PACKAGE` and rely on a proper dependency management setup instead. Katharsis comes out-of-the-box with a CDI and Spring integration, both implementing the `ServiceDiscovery` interface.
- make use of `JsonApiRelation` instead of `JsonApiToMany`, `JsonApiToOne`, `JsonApiLookupIncludeAutomatically` and `JsonApiIncludeByDefault`.

3.1 (soon)

- an experimental security module.
- a very experimental meta module.
- katharsis-ui

Requirements

Katharsis library requires minimum Java 7 to build and run.

Supported requests

Currently Katharsis covers most of the JSON API specification request types. The following table describes available requests that are currently accepted:

Table 1.1: A summary of the supported requests

HTTP method	Kind of request	Sample URL	Multiplicity of resource
GET	resources	http://host.local/tasks or http://host.local/tasks/1,2	multiple
	resource	http://host.local/tasks/1	single
	relationship	http://host.local/tasks/1/relationships/project	single
	field	http://host.local/tasks/1/project	single
POST	resource	http://host.local/tasks	single
	field	http://host.local/tasks/1/project	single
	relationship	http://host.local/tasks/1/relationships/project	single
PATCH	resource	http://host.local/tasks/1	single
	relationship	http://host.local/tasks/1/relationships/project	single
DELETE	resource	http://host.local/tasks/1	single
	relationship	http://host.local/tasks/1/relationships/project	single

Relationships

One of the main features of JSON API and Katharsis is support of managing relations between resources. To achieve that, two steps are required:

- Add a field annotated with JsonApiToOne or JsonApiToMany (depending on multiplicity of the relation) which will represent a unidirectional relation.
- Add a repository which defines operations that can be made on models.

Models

There are several annotations which can be assigned to models. By default all fields of the model are reflected in JSON API communication except synthetic fields. The annotations described below should be associated with either a field or a getter.

JsonApiResource

It is the most important annotation which defines a resource. It requires type parameter to be defined that is used to form a URLs and type field in passed JSONs. According to JSON API standard, the name defined in type can be either plural or singular

The example below shows a sample class which contains a definition of a resource.

```
@JsonApiResource(type = "tasks")
public class Task {
  // fields, getters and setters
}
```

JsonApiId

Defines a field which will be used as an identifier of a resource. Each resource requires this annotation to be present on a field which type implements `Serializable` or is of primitive type.

The example below shows a sample class which contains a definition of a field which contains an identifier.

```
@JsonApiResource(type = "tasks")
public class Task {
  @JsonApiId
  private Long id;

  // fields, getters and setters
}
```

JsonApiRelation

Indicates an association to either a single value or collection of resources. The type of such fields must be a valid resource.

The example below shows a sample class which contains this kind of relationship.

```
@JsonApiResource(type = "tasks")
public class Task {

  // ID field

  @JsonApiRelation(lookUp=LookupIncludeBehavior.AUTOMATICALLY_WHEN_NULL,
↪serialize=SerializeType.ONLY_ID)
  private Project project;

  // fields, getters and setters
}
```

The optional `serialize` parameter specifies how the association should be serialized when making a request. There are two things to consider. Whether related resources should be added to the `include` section of the response document. And whether the id of related resources should be serialized along with the resource in the corresponding `relationships.[name].data` section. Either `LAZY`, `ONLY_ID` or `EAGER` can be specified:

- `LAZY` only serializes the ID and does the inclusion if explicitly requested by the `include` URL parameter. This is the default.
- `ONLY_ID` always serializes the ID, but does only to an inclusion if explicitly requested by the `include` URL parameter.
- `EAGER` always both serializes the ID and does an inclusion.

There are two possibilities of how related resources are fetched. Either the requested repository directly returns related resources with the returned resources. Or Katharsis can take-over that work by doing nested calls to the corresponding `RelationshipRepositoryV2` implementations. The behavior is controlled by the optional `lookUp` parameter. There are three options:

- 'NONE' makes the requested repository responsible for returning related resources. This is the default.
- 'AUTOMATICALLY_WHEN_NULL' will let Katharsis lookup related resources if not already done by the requested repository.
- 'AUTOMATICALLY_ALWAYS' will force Katharsis to always lookup related resource regardless whether it is already done by the requested repository.

JsonApiMetaInformation

Field or getter annotated with `JsonApiMetaInformation` are marked to carry a `MetaInformation` implementation. See http://jsonapi.org/format/#document-meta for more information about meta data. Example:

```
@JsonApiResource(type = "projects")
public class Project {

        ...

        @JsonApiMetaInformation
        private ProjectMeta meta;

        public static class ProjectMeta implements MetaInformation {

                private String value;

                public String getValue() {
                        return value;
                }
                public void setValue(String value) {
                        this.value = value;
                }
        }
}
```

JsonApiLinksInformation

Field or getter annotated with `JsonApiLinksInformation` are marked to carry a `LinksInformation` implementation. See http://jsonapi.org/format/#document-links for more information about linking. Example:

```
@JsonApiResource(type = "projects")
public class Project {

        ...

        @JsonApiLinksInformation
        private ProjectLinks links;

        public static class ProjectLinks implements MetaInformation {

                private String value;

                public String getValue() {
                        return value;
                }

                public void setValue(String value) {
                        this.value = value;
                }
        }
}
```

Repositories

The modelled resources must be complemented by a corresponding repository implementation. This is achieved by implementing one of those two repository interfaces:

- ResourceRepositoryV2 for a resource
- RelationshipRepositoryV2 resp. BulkRelationshipRepositoryV2 for resource relationships

ResourceRepositoryV2

Base repository which is used to operate on resources. Each resource should have a corresponding repository implementation. It consist of five basic methods which provide a CRUD for a resource and two parameters: the first is a type of a resource and the second is a type of the resource's identifier.

The methods are as follows:

- `findOne(ID id, QuerySpec querySpec)` Search one resource with a given ID. If a resource cannot be found, a ResourceNotFoundException exception should be thrown. It should return an entity with associated relationships.

- `findAll(QuerySpec querySpec)` Search for all of the resources. An instance of QuerySpec can be used if necessary. If no resources can be found an empty Iterable or null must be returned. It should return entities with associated relationships.
- `findAll(Iterable<ID>ids, QuerySpec querySpec)` Search for resources constrained by a list of identifiers. An instance of QuerySpec can be used if necessary. If no resources can be found an empty Iterable or null must be returned. It should return entities with associated relationships.
- `save(S entity)` Saves a resource. It should not save relating relationships. A Returning resource must include assigned identifier created for the instance of resource. This method should be able to both create a new resource and update existing one.
- `delete(ID id)` Removes a resource identified by id parameter.

The ResourceRepositoryBase is a base class that takes care of some boiler-plate, like implementing findOne with findAll. An implementation can then look as simple as:

```
public class ProjectRepository extends ResourceRepositoryBase<Project, String> {

        private Map<Long, Project> projects = new HashMap<>();

        public ProjectRepository() {
                super(Project.class);
                save(new Project(1L, "Project A"));
                save(new Project(2L, "Project B"));
                save(new Project(3L, "Project C"));
        }

        @Override
        public synchronized void delete(String id) {
                projects.remove(id);
        }

        @Override
        public synchronized <S extends Project> S save(S project) {
                projects.put(project.getId(), project);
                return project;
        }

        @Override
        public synchronized ResourceList<Project> findAll(QuerySpec querySpec) {
                return querySpec.apply(projects.values());
        }
}
```

RelationshipRepositoryV2

Each relationship defined in Katharsis (annotation @JsonApiToOne and @JsonApiToMany) must have a relationship repository defined.

Base unidirectional repository responsible for operations on relations. All of the methods in this interface have fieldName field as their last parameter to solve the problem of many relationships between the same resources.

- `setRelation(T source, D_ID targetId, String fieldName)` Sets a resource defined by targetId to a field fieldName in an instance source. If no value is to be set, null value is passed.
- `setRelations(T source, Iterable<D_ID> targetIds, String fieldName)` Sets resources defined by targetIds to a field fieldName in an instance source. This is a all-or-nothing operation, that is no partial relationship updates are passed. If no values are to be set, empty Iterable is passed.

- `addRelations(T source, Iterable<D_ID> targetIds, String fieldName)` Adds relationships to a list of relationships.
- `removeRelations(T source, Iterable<D_ID> targetIds, String fieldName)` Removes relationships from a list of relationships.
- `findOneTarget(T_ID sourceId, String fieldName, QuerySpec querySpec)` Finds one field's value defined by fieldName in a source defined by sourceId.
- `findManyTargets(T_ID sourceId, String fieldName, QuerySpec querySpec)` Finds an Iterable of field's values defined by fieldName in a source defined by sourceId .

This interface must be implemented to let Katharsis work correctly, some of the requests are processed using only this kind of repository. As it can be seen above, there are two kinds of methods: for multiple and single relationships and it is possible to implement only one type of methods, e.g. singular methods. Nevertheless, it should be avoided because of potential future problems when adding new fields of other sizes.

In many cases, relationship operations can be mapped back to resource repository operations. Making the need for a custom relationship repository implementation redundant. A findManyTargets request might can be served by filtering the target repository. Or a relationship can be set by invoking the save operation on either the source or target resource repository (usually you want to save on the single-valued side). The ResourceRepositoryBase is a base class that takes care of exactly this. A repository implementation then looks as simple as:

```
public class ProjectToTaskRepository extends RelationshipRepositoryBase<Project, Long,
↪ Task, Long> {

        public ScheduleToTaskRepository() {
                super(Project.class, Task.class);
        }
}
```

For this to work, relations must be set up bidirectionally with the `opposite` attribute:

```
@JsonApiResource(type = "tasks")
public class Task {

        @JsonApiToOne(opposite = "tasks")
        @JsonApiIncludeByDefault
        private Project project;

    ...
}
```

BulkRelationshipRepositoryV2

BulkRelationshipRepositoryV2 extends RelationshipRepositoryV2 and provides an additional `findTargets` method. It allows to fetch a relation for multiple resources at once. It is recommended to make use of this implementation if a relationship is loaded frequently (either by a eager declaration or trough the `include` parameter) and it is costly to fetch that relation. RelationshipRepositoryBase provides a default implementation where `findOneTarget` and `findManyTargets` forward calls to the bulk `findTargets`.

ResourceList

ResourceRepositoryV2 and RelationshipRepositoryV2 return lists of type ResourceList. The ResourceList can carry, next to the actual resources, also meta and links information:

- **getLinks()** Gets the links information attached to this lists.
- **getMeta()** Gets the meta information attached to this lists.
- **getLinks(Class<L> linksClass)** Gets the links information of the given type attached to this lists. If the given type is not found, null is returned.
- **getMeta(Class<M> metaClass)** Gets the meta information of the given type attached to this lists. If the given type is not found, null is returned.

Thhere is a default implementation named DefaultResourceList. To gain type-safety, improved readability and katharsis-client support, application may provide a custom implementation extending ResourceListBase:

```
class ScheduleList extends ResourceListBase<Schedule, ScheduleListMeta,␣
↪ScheduleListLinks> {

}

class ScheduleListLinks implements LinksInformation {

        public String name = "value";

        ...
}

class ScheduleListMeta implements MetaInformation {

        public String name = "value";

        ...
}
```

This implementation can then be added to a repository interface declaration and used by both servers and clients:

```
public interface ScheduleRepository extends ResourceRepositoryV2<Schedule, Long> {

        @Override
        public ScheduleList findAll(QuerySpec querySpec);

}
```

Query parameters

Katharsis passes JSON API query parameters to repositories trough a QuerySpec parameter. It holds request parameters like sorting and filtering specified by JSON API. The subsequent sections will provide a number of example.

Filtering

Resource filtering can be achieved by providing parameters which start with `filter`. The format for filters: `filter[ResourceType][property|operator]([property|operator])* = "value"`

- `GET /tasks/?filter[name]=Super task`
- `GET /tasks/?filter[name][EQ]=Super task`
- `GET /tasks/?filter[tasks][name]=Super task`

- `GET /tasks/?filter[tasks][name]=Super task&filter[tasks][dueDate]=2015-10-01`

QuerySpec uses the `EQ` operator if no operator was provided. Custom operators can be registered with `DefaultQuerySpecDeserializer.addSupportedOperator(..)`. The default operator can be overridden by setting `DefaultQuerySpecDeserializer.setDefaultOperator(...)`.

Sorting

Sorting information for the resources can be achieved by providing `sort` parameter.

- `GET /tasks/?sort=name,-shortName`
- `GET /tasks/?sort[projects]=name,-shortName&include=projects`

Pagination

Pagination for the repositories can be achieved by providing `page` parameter. The format for pagination: `page[offset|limit] = "value", where value is an integer`

Example:

- `GET /tasks/?page[offset]=0&page[limit]=10`

Note that JSON API specifies first, previous, next and last links (see http://jsonapi.org/format/#fetching-pagination). Katharsis provides support to compute those pagination links. For this two work, a repository has to return meta and links information implementing PagedMetaInformation resp. PagedLinksInformation. With PagedMetaInformation the repository can let Katharsis know about the total number of (potentially filtered) resources. Katharsis then fills in PagedLinksInformation with the corresponding links.

Sparse Fieldsets

Information about fields to include in the response can be achieved by providing `fields` parameter.

- `GET /tasks/?fields=name`
- `GET /tasks/?fields[projects]=name,description&include=projects`

Inclusion of Related Resources

Information about relationships to include in the response can be achieved by providing `include` parameter. The format for fields: `include[ResourceType] = "property(.property)*"`

Examples:

- `GET /tasks/?include[tasks]=project`
- `GET /tasks/1/?include[tasks]=project`
- `GET /tasks/?include[tasks]=author`
- `GET /tasks/?include[tasks][]=author&include[tasks][]=comments`
- `GET /tasks/?include[projects]=task&include[tasks]=comments`
- `GET /tasks/?include[projects]=task&include=comments` (QuerySpec example)

DefaultQuerySpecDeserializer

Katharsis make use of `DefaultQuerySpecDeserializer` to map URL parameters to a QuerySpec instance. This instance is accessible from the various integrations, such as from the `KatharsisFeature`. It provides a number of customization options:

- **`setDefaultLimit(Long)`** Sets the page limit if none is specified by the request.
- **`setMaxPageLimit(Long)`** Sets the maximum page limit allowed to be requested.
- **`setIgnoreUnknownAttributes(boolean)`** DefaultQuerySpecDeserializer validates all passed parameters against the domain model and fails if one of the attributes is unknown. This flag allows to disable that check in case the should be necessary.

Note that appropriate page limits are vital to protect against denial-of-service attacks when working with large data sets!

`DefaultQuerySpecDeserializer` implements `QuerySpecDeserializer` and you may also provide your own implementation to further customize its behavior.

QuerySpec API

The API looks like (further setters available as well):

```
public class QuerySpec {
        public <T> List<T> apply(Iterable<T> resources){...}

        public Long getLimit() {...}

        public long getOffset() {...}

        public List<FilterSpec> getFilters() {...}

        public List<SortSpec> getSort() {...}

        public List<IncludeFieldSpec> getIncludedFields() {...}

        public List<IncludeRelationSpec> getIncludedRelations() {...}

        public QuerySpec getQuerySpec(Class<?> resourceClass) {...}

        ...
}
```

Note that single QuerySpec holds the parameters for a single resource type and, in more complex scenarios, request can lead to multiple QuerySpec instances (namely when related resources are also filtered, sorted, etc). A repository is invoked with the QuerySpec for the requested root type. If related resources are included in the request, their QuerySpecs can be obtained by calling `QuerySpec.getRelatedSpec(Class)` on the root QuerySpec.

> `FilterSpec` holds a value of type object. Since URL parameters are passed as String, they get converted to the proper types by the `DefaultQuerySpecDeserializer`. The type is determined based on the type of the filtered attribute.

QuerySpec provides a method `apply` that allows in-memory sorting, filtering and paging on any `java.util.Collection`. It is useful for testing and on smaller datasets to keep the implementation of a repository as simple as possible. It returns a ResourceList that carries a PagedMetaInformation that lets Katharsis automatically compute pagination links.

Error Handling

Processing errors in Katharsis can be handled by throwing an exception and providing a corresponding exception mapper which defines mapping to a proper JSON API error response.

Throwing an exception...

Here is an example of throwing an Exception in the code:

```
if (somethingWentWrong()) {
  throw new SampleException("errorId", "Oops! Something went wrong.")
}
```

Sample exception is nothing more than a simple runtime exception:

```
public class SampleException extends RuntimeException {

  private final String id;
  private final String title;

  public ExampleException(String id, String title) {
    this.id = id;
    this.title = title;
  }

  public String getId() {
    return id;
  }

  public String getTitle() {
    return title;
  }
}
```

...and mapping it to JSON API response

Class responsible for mapping the exception should:

- be annotated with ExceptionMapperProvider
- implement JsonApiExceptionMapper interface

Sample exception mapper:

```
@ExceptionMapperProvider
public class SampleExceptionMapper implements JsonApiExceptionMapper<SampleException>
↪{
  @Override
  public ErrorResponse toErrorResponse(SampleException exception) {
    return ErrorResponse.builder()
      .setStatus(HttpStatus.INTERNAL_SERVER_ERROR_500)
      .setSingleErrorData(ErrorData.builder()
        .setTitle(exception.getTitle())
        .setId(exception.getId())
        .build())
      .build();
    }
}
```

Exception mapper classes will be scanned for and registered during application startup. They should be located in your resource search package. If katharsis-cdi or katharsis-spring is used, the annotation is not necessary and it will instead be picked up through the dependency mechanisms.

An exception should be mapped to an ErrorResponse object. It consists of an HTTP status and ErrorData (which is consistent with JSON API error structure).

Note that the exception mapper is reponsible for providing the logging of exceptions with the appropriate log levels. Also have a look at the subsequent section about the validation module that takes care of JSR-303 bean validation exception mapping.

Meta Information

Note: With ResourceList and @JsonApiMetaInformation meta information can be returned directly. A MetaRepository implementation is no longer necessary.

There is a special interface which can be added to resource repositories to provide meta information: `io.katharsis.repository.MetaRepository`. It contains a single method `MetaInformation getMetaInformation(Iterable<T> resources)` which return meta information object that implements the marker `interface io.katharsis.response.MetaInformation`.

If you want to add meta information along with the responses, all repositories (those that implement `ResourceRepository` and `RelationshipRepository`) must implement `MetaRepository`.

When using annotated versions of repositories, a method that returns a `MetaInformation` object should be annotated with `JsonApiMeta` and the first parameter of the method must be a list of resources.

Links Information

Note: With ResourceList and @JsonApiLinksInformation links information can be returned directly. A LinksRepository implementation is no longer necessary.

There is a special interface which can be added to resource repositories to provide links information: `io.katharsis.repository.LinksRepository`. It contains a single method `LinksInformation getLinksInformation(Iterable<T> resources)` which return links information object that implements the marker `interface io.katharsis.response.LinksInformation`.

If you want to add meta information along with the responses, all repositories (those that implement `ResourceRepository` and `RelationshipRepository`), must implement `LinksRepository`.

When using annotated versions of repositories, a method that returns a `LinksInformation` object should be annotated with `JsonApiLinks` and the first parameter of the method has to be a list of resources.

JAX-RS integration

Katharsis allows integration with JAX-RS environments through the usage of JAX-RS specification. Under the hood there is a @PreMatching filter which checks each request for JSON API processing.

There are two ways to setup to integrate Katharsis into a JAX-RS application, depending on whether a dependency injection framework is in use. In either case the instantiated `KatharsisFeature` has to be registered as a JAX-RS feature.

Without Dependency Injection

Have a look at the Dropwizard example to see how to setup Katharsis without dependency injection.

Katharsis require an instance of every resources repository it finds. To provide them, `JsonServiceLocator` interface has to be implemented. The created instance of `JsonServiceLocator` has to be provided to new instance of `KatharsisFeature` along with Jackson Databind ObjectMapper.

```
@ApplicationPath("/")
public class MyApplication extends Application {

        @Override
        public Set<Object> getSingletons() {
                KatharsisFeature katharsisFeature = new KatharsisFeature(environment.
↪getObjectMapper(),
                new DefaultQuerySpecDeserializer(),
                new SampleJsonServiceLocator());
                return Collections.singleton((Object)katharsisFeature);
        }

        @Override
        public Map<String, Object> getProperties() {
                Map<String, Object> map = new HashMap<>();
                map.put(KatharsisProperties.RESOURCE_SEARCH_PACKAGE, "com.
↪myapplication.model")
                return map;
        }
}
```

In order for Katharsis to find its resources and repository, the `katharsis.config.core.resource.package` configuration property must be passed to JAX-RS. It allows configuring from which package should be searched to get models, repositories used by the core and exception mappers used to map thrown from repositories exceptions. Multiple packages can be passed by specifying a comma separated string of packages i.e. com.company.service.dto,com.company.service.repository.

With Dependency Injection (CDI, Spring)

The setup is simplified if a dependency injection framework like CDI or Spring is available. In this case, Katharsis can lookup its repositories, modules, etc. with that framework. To enable CDI support, add `io.katharsis:katharsis-cdi` to your classpath. Katharsis will then pickup the `CdiServiceDiscovery` implementation and use it to discover its modules and repositories. An application then looks as simple as:

```
@ApplicationPath("/")
public class WildflyApplication extends Application {

        @Override
        public Set<Class<?>> getClasses() {
                Set<Class<?>> set = new HashSet<>();
                set.add(KatharsisFeature.class);
                return set;
        }
}
```

Have a look at the wildfly example. The Spring setup follows the same pattern with a `SpringServiceDiscovery` and is explained in a subsequent section.

Providing a configuration

There are three parameters that can be passed to the server to get the configuration. All of them are defined in KatharsisProperties class:

- `katharsis.config.core.resource.domain`

  Domain name as well as protocol and optionally port number used when building links objects in responses i.e. http://katharsis.io. The value must not end with `/`. If the property is omitted, then they are extracted from the incoming request, which should work well for most use cases.

- `katharsis.config.web.path.prefix` (Optional)

  Default prefix of a URL path used in two cases:

  - When building `links` objects in responses
  - When performing method matching

  An example of a prefix `/api/v1`.

Customizing KatharsisFeature

`KatharsisFeature` has a number of constructors and methods that allow to customize its behavior. A more advanced setup may look like:

```
public class MyAdvancedKatharsisFeature implements Feature {

        @Inject
        private EntityManager em;

        @Inject
        private EntityManagerFactory emFactory;

        ...

        @Override
        public boolean configure(FeatureContext featureContext) {
                featureContext.property(KatharsisProperties.RESOURCE_SEARCH_PACKAGE, .
↪..);
                featureContext.property(KatharsisProperties.WEB_PATH_PREFIX, ...);

                // also map entities to JSON API resources (see further below)
                JpaModule jpaModule = new JpaModule(emFactory, em, transactionRunner);
                jpaModule.setRepositoryFactory(new ValidatedJpaRepositoryFactory());

                // JSON API compliant URL handling with QuerySpec
                DefaultQuerySpecDeserializer querySpecDeserializer = new␣
↪DefaultQuerySpecDeserializer();

                // limit all incoming requests to 20 resources if not specified␣
↪otherwise
                querySpecDeserializer.setDefaultLimit(20L);

                ServiceLocator serviceLocator = ...
                KatharsisFeature feature = new KatharsisFeature(new ObjectMapper(),␣
↪querySpecDeserializer, serviceLocator);
                feature.addModule(jpaModule);

                featureContext.register(feature);
                return true;
        }
}
```

Note that if the CDI or Spring integration is used, it will pickup any modules automatically.

Repository supported parameters

JAX-RS integration allows a developer to pass the following types of parameters in repository methods:

- An instance of `ContainerRequestContext`
- An instance of `SecurityContext`
- A cookie. The parameter should be annotated with `@CookieParam("cookie name")`. The type can be either `Cookie`, `String` or any other type that Jackson can handle.
- A header. The parameter should be annotated with `@HeaderParam("header name")`. The type can be either `String` or any other type that Jackson can handle.

Servlet integration

There are two ways of integrating katharsis using Servlets:

- Adding an instance of `AbstractKatharsisServlet`
- Adding an instance of `AbstractKatharsisFilter`

Integrating using a Servlet

To integrate Katharsis using a servlet several steps are required. The first one is to create a class that extends `AbstractKatharsisServlet` and will provide required configuration for the library. The code below shows a sample implementation:

```
import io.katharsis.invoker.KatharsisInvokerBuilder;
import io.katharsis.locator.JsonServiceLocator;
import io.katharsis.locator.SampleJsonServiceLocator;

import javax.servlet.ServletConfig;
import javax.servlet.ServletException;

public class SampleKatharsisServlet extends AbstractKatharsisServlet {
    private String resourceSearchPackage;
    private String resourceDefaultDomain;

    @Override
    public void init(ServletConfig servletConfig) throws ServletException {
        super.init(servletConfig);
        resourceSearchPackage = servletConfig
            .getInitParameter(KatharsisProperties.RESOURCE_SEARCH_PACKAGE);
        resourceDefaultDomain = servletConfig
            .getInitParameter(KatharsisProperties.RESOURCE_DEFAULT_DOMAIN);
    }

    /**
     * NOTE: A class extending this must provide a platform specific {@link␣
↪JsonServiceLocator}
     *       instead of the (testing-purpose) {@link SampleJsonServiceLocator} below
     *       in order to provide advanced dependency injections for the repositories.
     */
    @Override
    protected KatharsisInvokerBuilder createKatharsisInvokerBuilder() {
        return new KatharsisInvokerBuilder()
            .resourceSearchPackage(resourceSearchPackage)
            .resourceDefaultDomain(resourceDefaultDomain)
            .jsonServiceLocator(new SampleJsonServiceLocator());
    }

}
```

The newly created servlet must be added to the `web.xml` file or to another deployment descriptor. The code below shows a sample `web.xml` file with a properly defined and configured servlet:

```
<web-app>
  <servlet>
    <servlet-name>SampleKatharsisServlet</servlet-name>
    <servlet-class>io.katharsis.servlet.SampleKatharsisServlet</servlet-class>
    <init-param>
      <param-name>katharsis.config.core.resource.package</param-name>
      <param-value>io.katharsis.servlet.resource</param-value>
    </init-param>
    <init-param>
      <param-name>katharsis.config.core.resource.domain</param-name>
      <param-value>http://localhost:8080</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>SampleKatharsisServlet</servlet-name>
    <url-pattern>/api/v1/ *</url-pattern>
  </servlet-mapping>
</web-app>
```

Integrating using a filter

To integrate Katharsis using a filter, several steps are required. First, create a class that extends `AbstractKatharsisFilter`, which will provide required configuration for the library. The code below shows a sample implementation:

```java
import io.katharsis.invoker.KatharsisInvokerBuilder;
import io.katharsis.locator.JsonServiceLocator;
import io.katharsis.locator.SampleJsonServiceLocator;

import javax.servlet.FilterConfig;
import javax.servlet.ServletException;

public class SampleKatharsisFilter extends AbstractKatharsisFilter {

    private String resourceSearchPackage;
    private String resourceDefaultDomain;

    public void init(FilterConfig filterConfig) throws ServletException {
        super.init(filterConfig);
        resourceSearchPackage = filterConfig
            .getInitParameter(KatharsisProperties.RESOURCE_SEARCH_PACKAGE);
        resourceDefaultDomain = filterConfig
            .getInitParameter(KatharsisProperties.RESOURCE_DEFAULT_DOMAIN);
    }

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        super.init(filterConfig);
        resourceSearchPackage = filterConfig
            .getInitParameter(KatharsisProperties.RESOURCE_SEARCH_PACKAGE);
        resourceDefaultDomain = filterConfig
            .getInitParameter(KatharsisProperties.RESOURCE_DEFAULT_DOMAIN);
    }

    /**
     * NOTE: A class extending this must provide a platform specific {@link
↪JsonServiceLocator}
     *       instead of the (testing-purpose) {@link SampleJsonServiceLocator} below
     *       in order to provide advanced dependency injections for the repositories.
     */
    @Override
    protected KatharsisInvokerBuilder createKatharsisInvokerBuilder() {
        return new KatharsisInvokerBuilder()
            .resourceSearchPackage(resourceSearchPackage)
            .resourceDefaultDomain(resourceDefaultDomain)
            .jsonServiceLocator(new SampleJsonServiceLocator());
    }
}
```

The newly created filter must be added to `web.xml` file or other deployment descriptor. A code below shows a sample `web.xml` file with properly defined and configured filter

```xml
<web-app>
  <filter>
    <filter-name>SampleKatharsisFilter</filter-name>
    <filter-class>io.katharsis.servlet.SampleKatharsisFilter</filter-class>
    <init-param>
      <param-name>katharsis.config.web.path.prefix</param-name>
      <param-value>/api/v1</param-value>
    </init-param>
    <init-param>
      <param-name>katharsis.config.core.resource.package</param-name>
      <param-value>io.katharsis.servlet.resource</param-value>
    </init-param>
    <init-param>
      <param-name>katharsis.config.core.resource.domain</param-name>
      <param-value>http://localhost:8080</param-value>
    </init-param>
  </filter>
  <filter-mapping>
    <filter-name>SampleKatharsisFilter</filter-name>
    <url-pattern>/api/v1/ *</url-pattern>
  </filter-mapping>
</web-app>
```

Repository supported parameters

Servlet integration allows the following types of parameters in repository methods:

- An instance of `ServletContext`
- An instance of `HttpServletRequest`
- An instance of `HttpServletResponse`

Spring integration

Katharsis provides a simple Spring Boot integration using the `@Configuration` annotated class `KatharsisConfigV3`. Using this class, the only thing needed to allow Katharsis process requests is parameter configuration. An example `application.properties` file is presented below.

```
katharsis.domainName=http://localhost:8080
katharsis.pathPrefix=/api
```

Spring integration uses katharsis-servlet `AbstractKatharsisFilter` to fetch the requests. Similar to CDI, repositories and modules are picked up from the Spring ApplicationContext with `SpringServiceDiscovery`.

Repository supported parameters

Spring integration allows a developer to pass all of the types supported by Spring which don’t operate on the response.

Vertx integration

Katharsis provides `Handler` that intercepts requests and delegates them to Katharsis.

```
dependencies {
    compile 'io.katharsis:katharsis-vertx:<version>'
}
```

Simple usage example that creates the handler:

```
KatharsisHandler katharsisGlue = KatharsisHandlerFactory.create(Main.class.
↪getPackage().getName(), "/api");
router.route("/api/*").handler(katharsisGlue);
```

Advanced usage that shows how you can inject custom parameters in Katharsis repository methods:

```
ParameterProviderFactory factory = new SpringParameterProviderFactory(Json.mapper,␣
↪context);

KatharsisHandler katharsisGlue = KatharsisHandlerFactory.create(Main.class.
↪getPackage().getName(), "/api",
Json.mapper, new CustomParameterProviderFactory(Json.mapper, context));
router.route("/api/*").handler(katharsisGlue);
```

Client

There is a Katharsis client for Java projects to allow communicating with JSON-API compliant servers. Two http client libraries are supported:

- *OkHttp <http://square.github.io/okhttp>* Library has been used to allow usage in both Android and server applications and services.
- *Apache Http Client <https://hc.apache.org/httpcomponents-client-ga/index.html>* Library widely used in the Java community.

Add one of those library to the classpath and Katharsis will pick it up automatically.

The client requires to define resources in the same manner as defined in the *Models* section. To start using the client just create an instance of `KatharsisClient` and pass the service URL and the location to the package where the models are defined.

The client has three main methods:

- `KatharsisClient#getRepositoryForInterface(Class)` to obtain a resource repository stub from an existing repository interface.
- `KatharsisClient#getRepositoryForType(Class)` to obtain a generic resource repository stub from the provided resource type.
- `KatharsisClient#getRepositoryForType(Class, Class)` to obtain a generic relationship repository stub from the provided source and target resource types.

The interface of the repositories is as same as defined in *Repositories* section.

An example of the usage:

```
KatharsisClient client = new KatharsisClient("http://localhost:8080/api");
ResourceRepositoryV2<Task, Long> taskRepo = client.getRepositoryForType(Task.class);
List<Task> tasks = taskRepo.findAll(new QuerySpec(Task.class));
```

Have a look at, for example, the QuerySpecClientTest to see more examples of how it is used.

HTTP customization

It is possible to hook into the HTTP implementation used by Katharsis (OkHttp or Apache). Make use of `KatharsisClient#getHttpAdapter()` and cast it to either `HttpClientAdapter` or `OkHttpAdapter`. Both implementations provide a `addListener` method, which in turn gives access to the native builder used to construct the respective HTTP client implementation. This allows to cover various use cases:

- add custom request headers (security, tracing, etc.)
- collect statistics

- ...

You may have a look at `katharsis-brave` for an advanced example.

Modules

Katharsis has a module API that allows to extend the core functionality by third-party contributions. The mentioned JPA module in the next section is an example for that. The API is similar in spirit to the one of the `https://github.com/FasterXML/jackson`. The main interface is `Module` with a default implementation provided by `SimpleModule`. A module has access to a `ModuleContext` that allows to register all kinds of extensions like new `ResourceInformationBuilder`, `ResourceLookup`, `Filter`, `ExceptionMapper` and Jackson modules. It also gives access to the `ResourceRegistry` holding information about all the repositories registered to katharsis. The `JpaModule` in `katharsis-jpa` provides a good, more advanced example of using the module API.

Request Filtering

Katharsis provides three different, complementing mechanisms to hook into the request processing.

The `DocumentFilter` interface allows to intercept incoming requests and do any kind of validation, changes, monitoring, transaction handling, etc. `DocumentFilter` can be hooked into Katharsis by setting up a module and registering the filter to the `ModuleContext`. Not that for every request, this interface is called exactly once.

A request may span multiple repository accesses. To intercept the actual repository requests, implement the `RepositoryFilter` interface. `RepositoryFilter` has a number of methods that allow two intercept the repository request at different stages. Like `Filter` it can be hooked into Katharsis by setting up a module and registering the filter to the `ModuleContext`.

Similar to `RepositoryFilter` it is possible to decorate a repository with another repository implementing the same Katharsis repository interfaces. The decorated repository instead of the actual repository will get called and it is up to the decorated repository of how to proceed with the request, usually by calling the actual repository. `RepositoryDecoratorFactory` can be registered with `ModuleContext.addRepositoryDecoratorFactory`. The factory gets notified about every repository registration and is then free do decorate it or not.

Integrate third-party data stores

The core of Katharsis is quite flexible when it comes to implementing repositories. As such, it is not mandatory to make use of the Katharsis annotations and conventions. Instead, it is also (likely) possible to integrate an existing data store setup like JPA, JDBC, ElasticSearch, etc. into Katharsis. For this purpose a module can provide custom implementations of `ResourceInformationBuilder` and `RepositoryInformationBuilder` trough `ModuleContext.addResourceInformationBuilder` and `ModuleContext.addRepositoryInformationBuilder`. For example, the JpaModule of `katharsis-jpa` makes use of that to read JPA instead of Katharsis annotations. Such a module can then register additional (usually dynamic) repositories with `ModuleContext.addRepository`.

Integrate into a dependency injection environment

Katharsis comes with out-of-the-box support for Spring and CDI. Both of them implement `ServiceDiscovery`. You may provide your own implementation which can be hooked into the various Katharsis integrations, like the KatharsisFeature. Modules have access to that `ServiceDiscover` trough the `ModuleContext`.

Let a module hook into the Katharsis HTTP client implementation

Modules for the Katharsis client can additionally implement `HttpAdapterAware`. It gives the module access to the underlying HTTP client implementation and allows abitrary customizations of it. Have a look at the Katharsis client documentation for more information.

JPA Module

The JPA module allows to automatically expose JPA entities as JSON API repositories. No implementation or Katharsis-specific annotations are necessary.

The feature set includes:

- expose JPA entities to JSON API endpoints
- expose JPA relations as JSON API endpoints
- decide which entities to expose as endpoints
- sorting, filtering, paging, inclusion of related resources
- JPA filter API to modify the issued queries
- JPA Criteria API and QueryDSL support
- DTO mapping support
- support for computed attributes behaving like regular, persisted attributes.

JPA Setup

To use the module, add a dependency to `io.katharsis:katharsis-jpa` and register the `JpaModule` to Katharsis. For example in the case of JAX-RS:

```
TransactionRunner transactionRunner = ...;
JpaModule jpaModule = JpaModule.newServerModule(entityManagerFactory, entityManager,␣
↪transactionRunner);
jpaModule.setRepositoryFactory(new ValidatedJpaRepositoryFactory());

KatharsisFeature feature = new KatharsisFeature(...);
feature.addModule(jpaModule);
```

The JPA modules by default looks up the entityManagerFactory and obtains a list of registered JPA entities. For each entity a instance of `JpaEntityRepository` is registered to Katharsis using the module API. Accordingly, every relation is registered as `JpaRelationshipRepository`. `JpaModule.setRepositoryFactory` allows to provide a factory to change or customized the used repositories. To manually select the entities exposed to Katharsis use `JpaModule.addEntityClass(...)` and `JpaModule.removeEntityClass(...)`. If no `entityManagerFactory` is provided to newServerModule, then the registartion of entities is omitted and can be done manually.

The transactionRunner needs to be implemented by the application to hook into the transaction processing of the used environment (Spring, JEE, etc.). This might be as simple as a Spring bean implementing `TransactionRunner` and adding a `@Transactional` annotation. The JPA module makes sure that every call to a repository happens within such a transaction boundary.

To setup a Katharsis client with the JPA module use:

```
client = new KatharsisClient(getBaseUri().toString(), ...);

JpaModule module = JpaModule.newClientModule(TestEntity.class.getPackage().getName());
setupModule(module, false);
client.addModule(module);
```

The JpaModule takes care of the lookup of the entities and registering them to Katharsis with the provided package passed to `newClientModule`.

Have a look at https://github.com/katharsis-project/katharsis-framework/blob/develop/katharsis-jpa/src/test/java/io/katharsis/jpa/JpaQuerySpecEndToEndTest.java within the `katharsis-jpa` test cases to see how everything is used together with `katharsis-client`. The JPA modules further has a number of more advanced customization options that are discussed in the subsequent sections.

Criteria API and QueryDSL

The JPA module can work with two different query APIs, the default Criteria API and QueryDSL. `JpaModule.setQueryFactory` allows to choose between those two implementation. There is the `JpaCriteriaQueryFactory` and the `QuerydslQueryFactory`. By default the Criteria API is used. QueryDSL sits on top of JPQL and has to advantage of being easier to use.

Customizing the JPA repository

The setup page outlined the `JpaRepositoryFactory` that can be used to hook a custom JPA repository implementations into the JPA module. The JPA module further provides a more lightweight filter API to perform various changes to JPA repository requests:

`JpaModule.addFilter(new MyRepositoryFilter())`

A filter looks like:

```
public class MyRepositoryFilter extends JpaRepositoryFilterBase {

        boolean accept(Class<?> resourceType){...}

        <T, I extends Serializable> JpaEntityRepository<T, I>␣
↪filterCreation(JpaEntityRepository<T, I> repository){...}

        QuerySpec filterQuerySpec(Object repository, QuerySpec querySpec){...}

        ...
}
```

The various filter methods allow a wide variety of customizations or also to replace the passed object in question.

DTO Mapping

Mapping to DTO objects is supported with `JpaModule.registerMappedEntityClass(...)`. A mapper then can be provided that translates the Entity to a DTO class. Such a mapper might be implemented manually or generated (mostly) automatically with tools like MapStruct. If two mapped entities are registered, there respective mapped relationships will be automatically registered as well.

The mechanism is not limited to simple mappings, but can also introduce computed attributes like in the example depicted here:

```
JpaModule module = JpaModule.newServerModule(emFactory, em, transactionRunner);
                        module.setQueryFactory(QuerydslQueryFactory.newInstance());
QuerydslExpressionFactory<QTestEntity> basicComputedValueFactory = new␣
↪QuerydslExpressionFactory<QTestEntity>() {

        @Override
        public Expression<String> getExpression(QTestEntity parent, JPAQuery<?>␣
↪jpaQuery) {
                return parent.stringValue.upper();
        }
};

QuerydslQueryFactory queryFactory = (QuerydslQueryFactory) module.getQueryFactory();
queryFactory.registerComputedAttribute(TestEntity.class, TestDTO.ATTR_COMPUTED_UPPER_
↪STRING_VALUE,
         String.class, basicComputedValueFactory);
module.addMappedEntityClass(TestEntity.class, TestDTO.class, new␣
↪TestDTOMapper(entityManager));
```

and

```
public class TestDTOMapper implements JpaMapper<TestEntity, TestDTO> {

        @Override
        public TestDTO map(Tuple tuple) {
                TestDTO dto = new TestDTO();
                TestEntity entity = tuple.get(0, TestEntity.class);
                dto.setId(entity.getId());
                dto.setStringValue(entity.getStringValue());
                dto.setComputedUpperStringValue(tuple.get("computedUpperStringValue",␣
↪String.class));
                ...
                return dto;
        }

        ...

}
```

Some of the regular entity attributes are mapped to the DTO. But there is also a `computedUpperStringValue` attribute that is computed with an expression. The expression can be written with the Criteria API or QueryDSL depending on which query backend is in use.

Computed attributes are indistinguishable from regular, persisted entity attributes. They can be used for selection, sorting and filtering. Both `JpaCriteriaQueryFactory` and `QuerydslQueryFactory` provide a `registerComputedAttribute` method to register an expression factory to create such computed attributes. The registration requires the target entity and a name. To make the computed attribute available to consumers, the mapper class has access to it trough the provided tuple class. Have a look at https://github.com/katharsis-project/katharsis-framework/blob/develop/katharsis-jpa/src/test/java/io/katharsis/jpa/mapping/DtoMappingTest.java to see everything in use.

There is currently not yet any support for renaming of attribute. If attributes are renamed on DTOs, the incoming QuerySpec has to be modified accordingly to match again the entity attribute naming.

JSR 303 Validation Module

A `ValidationModule` provided by `io.katharsis:katharsis-validation` implements exception mappers for 'javax.validation.ValidationException' and 'javax.validation.ConstraintViolationException'. Among others, it properly translates 'javax.validation.ConstraintViolation' instances to JSON API errors. A JSON API error can, among others, contain a source pointer. This source pointer allows a clients/UI to display the validation errors next to the corresponding input fields.

Tracing with Zipkin/Brave

A `BraveModule` provided by `io.katharsis:katharsis-brave` provides integration into Zipkin/Brave to implement tracing for your repositories. The module is applicable to both a Katharsis client or server.

The Katharsis client can make use of either HttpClient or OkHttp to issue HTTP requests. Accordingly, a matching brave integration must be added to the classpath:

- `io.zipkin.brave:brave-okhttp`
- `io.zipkin.brave:brave-apache-http-interceptors`

The `BraveModule` then takes care of the integration and will create a client span for each request.

On the server-side, `BraveModule` creates a local span for each accessed repository. Every request triggers one or more repository accesses (depending on whether relations are included). Note however that `BraveModule` does not setup tracing for incoming requests. That is the purpose of the JAX-RS/servlet integration of Brave.

Security Module

This is an experimental module that intercepts all repository requests and performs Role-based access control. Have a look at the `SecurityModule` and the related `SecurityConfig` class. A setup can looks as follows:

```
Builder builder = SecurityConfig.builder();
builder.permitRole("allRole", ResourcePermission.ALL);
builder.permitRole("getRole", ResourcePermission.GET);
builder.permitRole("patchRole", ResourcePermission.PATCH);
builder.permitRole("postRole", ResourcePermission.POST);
builder.permitRole("deleteRole", ResourcePermission.DELETE);
builder.permitRole("taskRole", Task.class, ResourcePermission.ALL);
builder.permitRole("taskReadRole", Task.class, ResourcePermission.GET);
builder.permitRole("projectRole", Project.class, ResourcePermission.ALL);
builder.permitAll(ResourcePermission.GET);
builder.permitAll(Project.class, ResourcePermission.POST);
securityModule = SecurityModule.newServerModule(builder.build());
```

The security module further properly serializes javax.security authorization and authentication exceptions. As such it is also recommended to be used by KatharsisClient.

Meta Module

This is a (very) experimental module that exposes the internal workings of Katharsis as JSON API repositories. It lets you browse the set of available resources, their types, their attributes, etc. For example, Katharsis UI make use of the meta module to implement auto-completing of input fields. A setup can look as follows:

```
MetaModule metaModule = MetaModule.create();
metaModule.addMetaProvider(new ResourceMetaProvider());
```

To learn more about the set of available resources, have a look at the `MetaElement` class and all its subclasses, most notably `MetaResource` and `MetaResourceRepository`.

CHAPTER 2

How to improve or extend Katharsis

In this guide you will find out how to contribute to Katharsis.

NOTE: This is a work in progress.

We use gitter to chat with team members Katharsis Gitter .

If you have found a bug, please submit a patch to the repository. You'll find all of them on our Katharsis Github project.

I think Katharsis should do X?

If you have an idea for a new feature that Katharsis should support, it is a good idea to start a discussion about it on Katharsis Gitter or create a new issue on Katharsis Github .

If you also have code for the feature, definitely submit an issue and a pull request.

Please note that it might take some time for us to answer, since we are all volunteers.

How development happens

We use git flow to guide development. That means that branches have a special meaning and work is integrated using conventions defined by git flow.

We use `master` branch to keep the latest stable version.

We use `development` branch to gather new features until they are mature enough to make a release.

As described in git flow new features are developed in features branches. Once a feature is complete it should get merged in `development` via a `pull request` and `code review`.

In time, features gather and we can decide to make a new release. After a `feature freeze` period when only bug-fixes are allowed, we can release the new version of the project. We do this by tagging the last commit on `development`, building binaries, publishing artifacts to Maven Central and merging development into master to prepare for the next development cycle.

We use `2.x.y` style branches to keep maintenance versions. These version will not receive new features and are kept for bugfixing and creating maintenance releases.

CHAPTER 3

How to update katharsis.io

This guide explains how to update and deploy katharsis website.

Update

If you see a problem with the website, please submit an issue or better yet, submit a pull request to katharsis.io github repository.

Currently the website is an Angular application.

How to deploy changes

If you are a team member, you can deploy a new version of the website.

CHAPTER 4

Meeting notes

One of the main goal of the Katharsis Team is to support the community, that is not only provide support for submitted issues and answer questions on Gitter, but also conduct bi-weekly meetings where everyone can participate and discuss about Katharsis development and activities to make Katharsis better!

This page contains executive summary of each public meeting that had been held.

Meeting notes - 18.11.2016

- Lot of features have been added. Focus for the next release is improving documentation and sonar.
- katharsis-ui is next feature having top priority.

Meeting notes - 08.09.2016

- It should be noted that JPA is only for fast delivery models/projects
- katharsis-vertx should be updated to version 2.6.0
- reflections library should be removed from core, that is moved to integrations. Spring integration can make use of DI to get repositories and extract models from repository definitions
- When implementaing support for OPTIONS method, exceptions thrown from repository methods could be used to determine if a specific operation is supported
- Performane within Katharsis - try to find out bottlenecks, JMeter with flood.io might be a good choice
- katharsis-ui - JSON-API compilant browser actions: create a blank repo, sketch which libraries should be used, ask JSON-API team if someone would contribute

Meeting notes - 08.08.2016

- Create JPA integration into separate module
- Migration from Maven to Gradle can be postpone for later
- Possibilities and try to create the most suitable documentation tool and link it to the site. Consider GitBook as a possibility for keeping the documentation.
- Add automated deployment on travis from master branch that is tagged with version number
- Create contribution guide to creating issues and PRs
- Changelog should be removed from install step and move it to deploy step. Additionally release profile would be the most suitable for releasing process
- Let's try to release version (at least bug fixing) every month to provide concise development of library and limit scope of the changes.
- For Katharsis 3 release: We should start with #53 to allow users to deserialize the document to an appriopriate format e.g. a JSON with single resource should be represented as a single resource JSON object.
- Serialization and deserialization should be split into a seprate maven module to allow future katharsis-client implemtnation without any necessary dependencies to unused code
- We should focus on solving the bugs and make a bugfixing release till the end of 14.08

Meeting notes - 13.07.2016

- Framework integrations (e.g. katharsis-rs, katharsis-spring) should be merged with katharsis-core to allow easier building process
- Maven is good for Java projects, but it might be better switch to Gradle to improve project extendability and allow upload to bintray.com and Maven Central
- The documentation on the main page should redirect to readthedocs. All of the information from the current main page had been migrated except fancy graphics. Tutorials should also be included with not only about Katharsis but also JSON API. It could be good to make use of Apache projects template
- The development process should be well defined. As for now there are two processes, where the first for katharsis-vertex which uploads atrifacts to bintray.com and the second which uploads the rest of them. Since there is no Maven module configuration for Katharsis repositories, each project has to be uploaded separately.
- Deployment definitions should be removed from POMs
- Each week there should be one meeting and every one of them will end with a note
- There is a confusion about the purposes of Gitter rooms. katharsis-docs should include information what is the purpose of each room.