Crnk Documentation

Table of Contents

Crnk is a native resource-oriented rest library where resources, their relationships and repositories are the main building blocks. In that regard Crnk differ quite dramatically from most REST library out there and opens up many new possibilities. It allows you to rapidly build REST APIs without having to worry about lower protocol details and lets you instead focus on what matters: your application. On the HTTP/REST layer it follows the JSON API specification and recommendations. JSON API and Crnk come with support for:

- client and server implementation.
- standardized url handling such as `/api/persons?filter[title]=John` and `/api/persons/{id}`
- sorting, filtering, paging of resources
- attaching link and meta information to resources.
- inserting, updating and deleting of resources.
- support to request complex object graphs in a single request with JSON API inclusions.
- support for partial objects with sparse field sets.
- atomically create, update and delete multiple with jsonpatch.com.
- a flexible module API to choose and extend the feature set of Crnk.
- eased testing with the client implementation providing type-safe stubs to access server repositories.
- repositories providing runtime/meta information about Crnk to implement, for example, documentation and UI automation.
- generation of type-safe client stubs (currently Typescript as target language implemented)
- binding of Angular components with ngrx-json-api to Crnk endpoints.
- filters and decorates to intercept and modify all aspects of an application and Crnk.
- a module API to build and plugin-in reusable Crnk modules. Crnk comes with number of such (optional) modules to integrate with third-party libraries and frameworks.

Crnk is small, modular and lightweight. It integrates well with many popular frameworks and APIs:

- CDI: resolve repositories and extensions with CDI.
- Spring: run Crnk with Spring, including support for Spring Boot, ORM, Security and Sleuth.
- Reactor: for support of reactive programming.
- Servlet API: run Crnk as servlet.
- JAXRS: run Crnk as feature.
- JPA: expose entities as JSON API resources.
- bean validation: properly marshal validation and constraints exceptions.
- Zipkin, Brave, Spring Sleuth: trace all your calls.

While Crnk follows the JSON API specification, it is not limited to that. Have a look at the

http://www.crnk.io/[roadmap> for more information.

1. Examples

Crnk comes with various examples. There is a main example application in a dedicated repository available from crnk-example. It **shows an end-to-end example** with Crnk, Angular, Spring Boot and ngrx-json-api.

And there are various simpler example applications that show the integration of Crnk into various frameworks like:

- `spring-boot-example`
- `wildfly-example`
- `dropwizard-mongo-example`
- `dropwizard-simple-example`
- `jersey-example`
- `dagger-vertx-example` showcasing a very lightweight setup with Dagger, Vert.x, Proguard, OpenJ9 VM having a small size, startup time and memory footprint.

available from crnk-examples.

The impatient may also directly want to jump to ResourceRepositoryV2, but it is highly recommended to familiarize one self with the architecture and annotations as well. Unlike traditional REST libraries, Crnk comes with a lot of built-in semantics that allow to automate otherwise laborious tasks.

2. Architecture

Resources, **relationships** and **repositories** are the central building blocks of Crnk:

- Resources hold data as value fields, meta information and link information.
- Relationships establish links between resources.
- **resource repositories** and **relationship repositories** implement access to resources and relationships.

A Crnk application models its API as resources and relationships. It is not uncommon for applications to also have a few remaining service-oriented APIs. Later chapters will show how Crnk integrates with other libraries like JAX-RS or Spring MVC to achieve this. Based on such a model, an application implements repositories to provide access trough that model.

Currently implemented is the JSON API specification to access that model as part of the `crnk-core` project. The JSON API specification provide all the essential building blocks like sorting, filtering, paging, document formats, linking and error handling to access resources and relationships. But

other implementations, such as GraphQL or different kinds of REST/JSON APIs, are possible as well. With JSON API, an incoming request is processed as follows:

- A Crnk interceptor is called from the underlying framework. This might be, for example, from a Servlet environment, JAX-RS or Spring MVC.
- The request is deserialized to Crnk data structures like `Document`, `Resource`, `ResourceIdentifier` or `ErrorData`.
- The type of request is determined: whether it is a `POST`, `PATCH`, `GET` or `DELETE` request and whether it is a resource or relationship request.
- The request is forwarded to the appropriate repository.
- `GET` requests can ask for inclusions of further, related resources. Result resources will then trigger further requests to other repositories. This can happen either manually from within the initially called repository or automatically by Crnk (explained in detail in later chapters).
- The result resources are merged into response document and returned to the underlying framework for delivery. Possible exceptions are handled as and mapped as well.

A benefit of Crnk is its flexibility how to set all this up:

- Resources and relationships can be defined with simple Java Beans and annotations or programmatically. The later allows virtually any kind of customization at runtime, like setting up repositories dynamically. One example is `crnk-jpa` that is able to expose any JPA entity as JSON API resource.
- Resources and relationships can be entirely decoupled concerns. For example, an independent relationship repository `C` can introduce a relationship between a resource `a` and resource `b` implemented by resource repositories `A` and `B`. For example, an audit component could intercept and log any modifications. Access to it is provided by introducing a new relationship `history` onto each resource.
- Information about resources, relationships and repositories are available trough a Java API and JSON API endpoint.
- Filters and decorators allow to intercept and modify every step along the request chain. This can be used, for example, to enforce security, collect metrics or do tracing.

To facilitate the setup, Crnk comes with a small module API. Independent functionality can be assembled as module and then just included into the application. Crnk comes with a number of modules on its own:

- `crnk-jpa`
- `crnk-validation`
- `crnk-operations`
- various Spring modules
- ...

Such modules can make use of filters, decorators, decoupled resources and relationships and various other features. Everything together fosters the use of the **composite pattern** where larger applications can be assembled from smaller parts, some from third-party modules and others

manually implemented.

The part of `crnk-core` taking care of all this is denoted as the `engine`. The subsequent chapters explain how to setup and use Crnk.

3. Setup

Crnk integrates well with many popular framework. The example applications outline various different possible setups. But application are also free to customize their setup to their liking. There are three main, orthogonal aspects of Crnk that need configuration:

1. The integration into a web framework like JAXRS or the Servlet API to be able to process requests.
2. The discovery of repositories, modules, exception mappers, etc. Usually by a dependency injection framework. But can also happen manually.
3. The selection of third-party modules to reuse. For a list of modules provided by Crnk see the <modules> chapter.

The subsequent sections explain various possibilities resp. how to implement a custom one. The [reactive] chapter further outlines how to setup Crnk in an asynchronous/reactive setting.

3.1. Requirements

Crnk library requires minimum Java 8 (as of Crnk 2.4) to build and run. In the future it will come with support for both the current major Java releases (9, 10, 11, etc.) and the current long-term support version that gets released every three years.

3.2. BOM

With `io.crnk:crnk-bom` a Maven BOM is provided that manages the dependencies of all crnk artifacts. In Gradle the setup then looks as follows:

```
buildscript {
    dependencies {
        classpath "io.spring.gradle:dependency-management-plugin:1.0.4.RELEASE"
    }
}

gradle.beforeProject { Project project ->
    project.with {
        apply plugin: 'io.spring.dependency-management'
        dependencyManagement {
            imports {
                mavenBom "io.crnk:crnk-bom:$CRNK_VERSION"
            }
        }
    }
}
```

The crnk modules can then simply be used without having to specify a version:

```
dependencies {
    compile 'io.crnk:crnk-rs'
    compile 'io.crnk:crnk-setup-spring-boot2'
    ...
}
```

3.3. Logging

Crnk makes use of SLF4J to do logging. Make sure to have the API properly setup. For example by making use of Logback or one of the many bridges to other Logging frameworks.

TIP Set `io.crnk` to `DEBUG` if you encounter any issues during setup or later at runtime.

3.4. Integration with JAX-RS

Crnk allows integration with JAX-RS environments through the usage of JAX-RS specification. JAX-RS 2.0 is required for this integration. Under the hood there is a @PreMatching filter which checks each request for JSON API processing. The setup can look as simple as:

3.4.1. CrnkFeature

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

        @Override
        public Set<Object> getSingletons() {
            CrnkFeature crnkFeature = new CrnkFeature();
            return Collections.singleton((Object)crnkFeature);
        }
    }
```

`CrnkFeature` provides various accessors to customize the behavior of Crnk. A more advanced setup may look like:

```
    public class MyAdvancedCrnkFeature implements Feature {

        @Inject
        private EntityManager em;

        @Inject
        private EntityManagerFactory emFactory;

        ...

        @Override
        public boolean configure(FeatureContext featureContext) {
            // also map entities to JSON API resources (see further below)
            JpaModule jpaModule = new JpaModule(emFactory, em, transactionRunner);
            jpaModule.setRepositoryFactory(new ValidatedJpaRepositoryFactory());

            // limit all incoming requests to 20 resources if not specified otherwise
            DefaultQuerySpecUrlMapper urlMapper = new DefaultQuerySpecUrlMapper();
            urlMapper.setDefaultLimit(20L);

            ServiceLocator serviceLocator = ...
            CrnkFeature feature = new CrnkFeature();
            feature.addModule(jpaModule);
            feature.getBoot().setUrlMapper(urlMapper);

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

Crnk will install a JAX-RS filter that will intercept and process any Crnk-related request.

Note that depending on the discovery mechanism in use (like Spring or CDI), modules like this JpaModule can be picked up automatically and do not manual registration.

3.4.2. Exception mapping for JAX-RS services

In many cases Crnk repositories are used along regular JAX-RS services. In such scenarios it can be worthwhile if Crnk repositories and JAX-RS services make use of the same exception handling and response format. To make use of the JSON API resp. Crnk exception handling in JAX-RS services, one can add the **JsonapiExceptionMapperBridge** to the JAX-RS application. The constructor of `JsonapiExceptionMapperBridge` takes `CrnkFeature` as parameter.

For an example have a look at the next section which make use of it together with `JsonApiResponseFilter`.

3.4.3. Use JSON API format with JAX-RS services

Similar to `JsonapiExceptionMapperBridge` in the previous section, it is possible for JAX-RS services to return resources in JSON API format with **JsonApiResponseFilter**. `JsonApiResponseFilter` wraps primitive responses with a `data` object; resource objects with `data` and `included` objects. The constructor of `JsonApiResponseFilter` takes `CrnkFeature` as parameter.

To determine which JAX-RS services should be wrapped, `JsonApiResponseFilter` checks whether the `@Produce` annotation delivers JSON API. The produce annotation can be added, for example, to the class:

ScheduleRepository.java

```
@Path("schedules")
@Produces(HttpHeaders.JSONAPI_CONTENT_TYPE)
```

And the JAX-RS application setup looks like:

JsonApiResponseFilterTestBase.java

```
    @ApplicationPath("/")
    class TestApplication extends ResourceConfig {

        TestApplication(JsonApiResponseFilterTestBase instance, boolean
enableNullResponse) {
            instance.setEnableNullResponse(enableNullResponse);

            property(CrnkProperties.RESOURCE_SEARCH_PACKAGE, "io.crnk.rs.resource");
            property(CrnkProperties.NULL_DATA_RESPONSE_ENABLED,
Boolean.toString(enableNullResponse));

            CrnkFeature feature = new CrnkFeature();
            feature.addModule(new TestModule());

            register(new JsonApiResponseFilter(feature));
            register(new JsonapiExceptionMapperBridge(feature));
            register(new JacksonFeature());

            register(feature);
        }
    }
```

Note that:

- `CrnkProperties.NULL_DATA_RESPONSE_ENABLED` determines whether null responses should be wrapped as JSON API responses.
- Make use of proper service discovery instead of `CrnkProperties.RESOURCE_SEARCH_PACKAGE` in real applications.

3.4.4. JAX-RS service interoperability

It is possible to implement repositories that host both JAX-RS and JSON-API methods to complement JSON API repositories with non-resource based services. Have a look at the Crnk Client chapter for an example.

3.5. Integration with Servlet API

There are two ways of integrating crnk using Servlets:

- Adding an instance of `CrnkServlet`
- Adding an instance of `CrnkFilter`

3.5.1. Integrating using a Servlet

There is a `CrnkServlet` implementation allowing to integrate Crnk into a Servlet environment. It can be configured with all the parameters outlined in the subsequent sections. Many times application

will desire to do more advanced customizations, in this case one can extends `CrnkServlet` and get access to `CrnkBoot`. The code below shows a sample implementation:

SampleCrnkServlet.java

```
public class SampleCrnkServlet extends CrnkServlet {

    @Override
    protected void initCrnk(CrnkBoot boot) {
        // do your configuration here
    }
}
```

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>SampleCrnkServlet</servlet-name>
      <servlet-class>io.crnk.servlet.SampleCrnkServlet</servlet-class>
      <init-param>
        <!-- can typically be ommitted and is auto-detected -->
        <param-name>crnk.config.core.resource.domain</param-name>
        <param-value>http://www.mydomain.com</param-value>
      </init-param>
    </servlet>
    <servlet-mapping>
      <servlet-name>SampleCrnkServlet</servlet-name>
      <url-pattern>/api/v1/ *</url-pattern>
    </servlet-mapping>
  </web-app>
```

`init-param` allow to pass configuration flags to Crnk. For a list of properties see here.

3.5.2. Integrating using a filter

Integrating Crnk as a Servlet filter works in a very similar fashion as for servlets:

SampleCrnkFilter.java

```
public class SampleCrnkFilter extends CrnkFilter {

    @Override
    protected void initCrnk(CrnkBoot boot) {
        // do your configuration here
    }
}
```

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

```
<web-app>
  <filter>
    <filter-name>SampleCrnkFilter</filter-name>
    <filter-class>io.crnk.servlet.SampleCrnkFilter</filter-class>
    <init-param>
      <!-- can typically be ommitted and is auto-detected -->
      <param-name>crnk.config.core.resource.domain</param-name>
      <param-value>http://www.mydomain.com</param-value>
    </init-param>
  </filter>
</web-app>
```

`init-param` allow to pass configuration flags to Crnk. For a list of properties see here.

3.6. Integration with Spring and String Boot

Crnk provides with:

- `io-crnk:crnk-setup-spring` support for plain Spring 4 and 5.
- `io-crnk:crnk-setup-spring-boot1` support for Spring Boot 1.x. This module is considered being deprecated and will be removed in the future.
- `io-crnk:crnk-setup-spring-boot2` support for Spring Boot 2.x.

There is a `CrnkCoreAutoConfiguration` in `crnk-setup-spring-boot2` that outlines the basic setup that can easily be applied to a Spring-only setup without Spring Boot using `crnk-setup-spring`:

- It uses the `CrnkFilter` servlet filter to fetch the requests.
- Service discovery is performed with `SpringServiceDiscovery` using the Spring `ApplicationContext`.

There are further auto configurations that automatically setup various Crnk modules. And there are also further integrations into Spring (like for `RestTemplate` and Spring Security). More information is available in Spring modules.

The web integration can be customized for Spring Boot setups with the subsequent properties:

application.properties

```
crnk.enabled=true
crnk.domain-name=http://localhost:8080
crnk.path-prefix=/api
crnk.default-page-limit=20
crnk.max-page-limit=1000
crnk.allow-unknown-attributes=false
crnk.return404-on-null=true
```

See CrnkCoreProperties for more information. The Spring modules and their auto configurations make further configuration properties available.

3.7. Integration with Vert.x

CAUTION	Reactive programming support has been introduced in Crnk 2.6 and is still considered experimental with some limitations. Please also provide feedback about this Vert.x integration.

Crnk integrates with Vert.x RxJava 2 using `crnk-reactive` and `crnk-setup-vertx`. More information about reactive programming is available here. To make use of Crnk with Vert.x, make sure you have the following dependencies specified:

```
    compile 'io.crnk:crnk-setup-vertx'
    compile 'io.vertx:vertx-rx-java2'
```

An example Vert.x vehicle may then look like:

CrnkVerticle.java

```
public class CrnkVerticle extends AbstractVerticle {

    private static final Logger LOGGER = LoggerFactory.getLogger(CrnkVerticle.class);

    public ReactiveTestModule testModule = new ReactiveTestModule();

    private int port;

    public CrnkVerticle(int port) {
        this.port = port;
    }

    @Override
    public void start() {
        HttpServer server = vertx.createHttpServer();

        CrnkVertxHandler handler = new CrnkVertxHandler((boot) -> {
            boot.addModule(HomeModule.create());
            boot.addModule(testModule);
        });

        server.requestStream().toFlowable()
                .flatMap(request -> handler.process(request))
                .subscribe((response) -> LOGGER.debug("delivered response {}",
response), error -> LOGGER.debug("error occured", error));
        server.listen(port);
    }
}
```

`CrnkVertxHandler` holds the Crnk setup. Its constructor takes a `Consumer<CrnkBoot>` that allows the customization of Crnk. The example makes use of it to register two modules. `CrnkVertxHandler.process` is the main method that allows to process `HttpServerRequest` objects of Vert.x.

3.8. Discovery with CDI

To enable CDI support, add `io.crnk:crnk-cdi` to your classpath. Crnk will then pickup the `CdiServiceDiscovery` implementation and use it to discover its modules and repositories. Modules, repositories, etc. will then be picked up if they are registered as CDI beans.

By default `Cdi.current()` is used to obtain a `BeanManager`. The application may also make use of `CdiServiceDiscovery.setBeanManager(…)` to set a custom one. The various integrations like `CrnkFeature` provide a `setServiceDiscovery` method to set a customized instance.

WARNING	`Cdi.current()` has shown to be unreliable in some cases when doing EAR deployment. In such cases it is highly recommended to set the `BeanManager` manually.

3.9. Discovery with Guice

A `GuiceServiceDiscovery` implementation is provided. The various integrations like `CrnkFeature` provide a `setServiceDiscovery` method to set the instance. For an example have a look at the dropwizard example application (https://github.com/crnk-project/crnk-framework/tree/master/crnk-examples/dropwizard-simple-example).

3.10. Discovery with Spring

The Spring integration comes with a `SpringServiceDiscovery` that makes use of the Spring `ApplicationContext` to discover beans.

3.11. Discovery without a dependency injection framework

WARNING	This mechanism is considered to be rather deprecated and a lightweight dependency injection framework like guice or the setup of manual modules is recommended instead.

If no dependency injection framework is used, Crnk can also discover beans on its own. For this purpose, the `org.reflections:reflections` library has to be added to the classpath and the `CrnkProperties.RESOURCE_SEARCH_PACKAGE` be defined. In JAX-RS this may look like:

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

        @Override
        public Set<Object> getSingletons() {
            CrnkFeature crnkFeature = new CrnkFeature();
            crnkFeature.getBoot().setServiceLocator(...);
            return Collections.singleton((Object)crnkFeature);
        }

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

A `JsonServiceLocator` service locator can be provided to control the instatiation of object. By default the default constructor will be used. The `CrnkProperties.RESOURCE_SEARCH_PACKAGE` property is passed to define which package should be searched for beans. Multiple packages can be passed by specifying a comma separated string of packages i.e. com.company.service.dto,com.company.service.repository. It will pick up any public non-abstract

class that makes use of Crnk interfaces, like repositories, exception mappers and modules.

3.12. No Discovery

It is also possible to make use of no discovery mechanism at all. In this case it is still possible to add repositories and other features through modules. A simple example looks like:

DropwizardService

```
        SimpleModule module = new SimpleModule("example");
        module.addRepository(new ProjectRepository());

        CrnkFeature crnkFeature = new CrnkFeature();
        crnkFeature.addModule(module);

        environment.jersey().register(crnkFeature);
```

Have a look at the various [modules] chapters for more information.

3.13. Implement a custom discovery mechanism

Application can bring along there own implementation of `ServiceDiscovery`. For more information see here.

3.14. CrnkBoot

`CrnkBoot` is a class shared among all the different integrations that takes care of setting up and starting Crnk. Every integration will provide access to it, which in turn allows for virtually any kind of customization.

Some possibilities:

- `getObjectMapper` allows access to the used Jackson instance.
- `addModule` allows to add a module.
- `setServiceDiscovery` sets a custom service discovery mechanism.
- `setPropertiesProvider` allows to set how properties are resolved.
- `getQuerySpecDeserializer` and `setQuerySpecDeserializer` allows to reconfigure how parameters are parsed. Note that in some areas JSON API only provides reocmmendations and Crnk follows those recommendations by default. So depending on your use cases, you may want to configure or implement some aspects differently.
- `setMaxPageLimit` allows to set the maximum number of allowed resources that can be fetched with a request by limiting pagination.
- `setDefaultPageLimit` allows to set a default page limit if none is specified by the request. **Highly**

recommended to be used as people frequently browse repositories on there own with a web browser and fail to provide pagination. As a result, your entire database may get downloaded and may bring down your servers depending on the datasize.

IMPORTANT	Appropriate page limits are vital to protect against denial-of-service attacks when working with large data sets! Such attacks may not be of malicious nature, but normal users using a browser and just omitting to specify pagination parameters.

3.15. Properties

Any of the integrations allows API access to customize Crnk. There are also a number of configuration flags provided by `CrnkProperties`:

- `crnk.config.core.resource.domain` Domain name as well as protocol and optionally port number used when building links objects in responses i.e. http://crnk.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.
- `crnk.config.web.path.prefix` 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`.
- `crnk.config.include.paging.packagingEnabled` enables pagination for inclusions. Disabled by default. Be aware this may inadvertently enable pagination for included resources when doing paging on root resources if data structures are cyclic. See `CrnkProperties.INCLUDE_PAGING_ENABLED` fore mor information.
- `crnk.config.include.behavior` with possible values `BY_TYPE` (default) and `BY_ROOT_PATH`. `BY_ROOT_PATH` specifies that an inclusion can only requested as path from the root resource such as `include[tasks]=project.schedule`. While `BY_TYPE` can further request inclusions by type directly such as `include[tasks]=project&include[projects]=schedule`. For simple object structures they are semantically the same, but they do differ for more complex ones, like when multiple attributes lead to the same type or for cycle structures. In the later case BY_TYPE inclusions become recursive, while BY_ROOT_PATH do not. Note that the use of BY_TYPE outmatches BY_ROOT_PATH, so BY_TYPE includes everything BY_ROOT_PATH does and potentially more. For more information see `CrnkProperties.INCLUDE_BEHAVIOR`.
- `crnk.config.resource.immutableWrite` with values IGNORE (default) or FAIL. Determines how to deal with field that cannot be changed upon a PATCH or POST request. For more information see `CrnkProperties.RESOURCE_FIELD_IMMUTABLE_WRITE_BEHAVIOR`.
- `crnk.config.resource.response.return_404` with values true and false (default). Enforces a 404 response should a repository return a null value. This is common practice, but not strictly mandated by the JSON API specification. In general it is recommended for repository to throw `ResourceNotFoundException`.
- `crnk.config.serialize.object.links` to serialize links as objects. See http://jsonapi.org/format/#

document-links. Disabled by default.

- `crnk.config.resource.request.rejectPlainJson` whether to reject GET requests with `application/json` accept headers and enforce `application/vnd.api+json`. Disabled by default.
- `crnk.config.resource.request.allowUnknownAttributes` lets Crnk ignore unknown filter and sort parameters. Disabled by default.
- `crnk.config.serialize.object.links` determines whether links should be serialized as simple string (default) or as objects (with a `self` attribute holding the url).
- `crnk.config.resource.request.rejectPlainJson` determines whether Crnk should reject `application/json` requests to JSON-API endpoints. Disabled by default. The JSON-API specification mandates the use of the `application/vnd.api+json` MIME-Type. In cases where frontends or intermediate proxies prefer the `application/json` MIME-Type, that type can be sent in the `Accept` header instead. If an application wants to serve a different response depending on whether the client's `Accept` header contains `application/vnd.api+json` or `application/json`, this option can be enabled. This *does not affect the payload `Content-Type`. `POST` and `PATCH` requests must still use `Content-Type: application/vnd.api+json` to describe their request body

4. Resource

A resource as defined by JSON API holds the actual data. The engine part of `crnk-core` is agnostic to how such resources are actually implemented (see the [architecture] and [modules] chapters). This chapter describes the most common way Java Beans and annotations. See here for more information how setup resources and repositories programmatically at runtime.

4.1. 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
}
```

where `type` parameter specifies the resource's name.

The optional `pagingBehavior` parameter allows to set a pagination behavior of the resource. In order to do it you have to specify a type which is actually an implementation of a `PagingBehavior` interface.

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

More detailed information about pagination behaviors could be found at Pagination section.

4.2. 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
  }
```

4.3. 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=Serial
izeType.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 Crnk 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 Crnk lookup related resources if not already done by the requested repository.
- `AUTOMATICALLY_ALWAYS` will force Crnk to always lookup related resource regardless whether it is already done by the requested repository.

There are many different ways how a relationship may end-up being implemented. In the best case, no implementation is necessary at all and requests can be dispatches to one of the two related resource repositories. The `repositoryBehavior` allows to configure behavior:

- `DEFAULT` makes use of `IMPLICIT_FROM_OWNER` if a relationship also makes use of `@JsonApiRelationId` (see below) or `lookUp=NONE` (see above). In any other case it expects a custom implementation.
- `CUSTOM` expects a custom implementation.
- `FORWARD_OWNER` forward any relationship request to the owning resource repository, the repository that defines the requested relationship field. GET requests will fetch the owning resources and grab the related resources from there (with the appropriate inclusion parameter). This assumes that the owning resource properties hold the related resources (or at least there IDs in case of `JsonApiRelationId`, see below). POST, PATCH, DELETE requests will update the properties of the owning resource accordingly and invoke a save operation on the owning resource repository. An implementation is provided by `ImplicitOwnerBasedRelationshipRepository`.
- `FORWARD_GET_OPPOSITE_SET_OWNER` works like `FORWARD_OWNER` for PATCH, POST, DELETE methods. In contrast, GET requests are forwarded to the opposite resource repository. For example, if there is a relationship between `Task` and `Project` with the `project` and `tasks` relationship fields. To get all tasks of a project, the task repository will be queried with a `project.id=<projectId>` filter parameter. Relational database are one typical example where this pattern fits nicely. In contract to `IMPLICIT_FROM_OWNER` only a single resource repository is involved with a slightly more complex filter parameter, giving performance benefits. An implementation is provided by `RelationshipRepositoryBase`.

- `FORWARD_OPPOSITE` the opposite to `FORWARD_OWNER`. Querying works like `IMPLICIT_GET_OPPOSITE_MODIFY_OWNER`.

The forwarding behaviors are implemented by `ForwardingRelationshipRepository`.

IMPORTANT	It likely takes a moment to familiarize oneself with all configuration options of `@JsonApiRelation` and the subsequent `@JsonApiRelationId`. But at the same time it is one area where a native resource-oriented REST library like Crnk can provide significant benefit and reduce manual work compared to more classical REST libraries like Spring MVC or JAX-RS.

4.4. JsonApiRelationId

Fields annotated with `@JsonApiRelation` hold fully-realized related resources. There are situations where the id of a related resource is available for free or can be obtained much more cheaply then fetching the entire related resource. In this case resources can make use of fields annotated with `@JsonApiRelationId`. The complement `@JsonApiRelation` fields by holding there ID only. An example looks like:

```
@JsonApiResource(type = "schedules")
public class Schedule {
  ...

  @JsonApiRelationId
  private Long projectId;

  @JsonApiRelation
  private Project project;

  public Long getProjectId() {
    return projectId;
  }

  public void setProjectId(Long projectId) {
    this.projectId = projectId;
    this.project = null;
  }

  public Project getProject() {
    return project;
  }

  public void setProject(Project project) {
    this.projectId = project != null ? project.getId() : null;
    this.project = project;
  }
}
```

Notice that: - `Schedule` resource holds both a `project` and `projectId` field that point to the same related resource. - setters must set both properties to make sure they stay in sync. If only the ID is set, the object must be nulled. - `propertyId` will never show in requests and responses. It can be considered to be `transient`.

By default, the naming convention for `@JsonApiRelationId` field is to end with a `Id` or `Ids` suffix. Crnk will the pair those two objects automatically. Trailing `s` are ignored for multi-valued fields, meaning that `projectIds` matches with `projects`. But it is also possible to specify a custom name, for example:

```
@JsonApiRelationId
private Long projectFk;

@JsonApiRelation(idField = "projectFk")
private Project project;
```

If a `@JsonApiRelationId` field cannot be matched to a `@JsonApiRelation` field, an exception will be thrown.

`@JsonApiRelationId` fields are used for:

- `GET` requests to fill-in the `data` section of a relationship.
- `POST` and `PATCH` requests to fill-in the new value without having to fetch and set the entire related resource.

Further (substantial) benefit for `@JsonApiRelationId` fields is that no `RelationshipRepository` must be implemented. Instead Crnk will automatically dispatch relationship requests to the owning and opposite `ResourceRepository`. This allows to focus on the development of `ResourceRepository`. See RelationshipRepository for more information.

4.5. 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;
        }
    }
}
```

4.6. 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 LinksInformation {

            private String value;

            public String getValue() {
                return value;
            }

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

By default links are serialized as:

```
"links": {
  "self": "http://example.com/posts"
}
```

With `crnk.config.serialize.object.links=true` links get serialized as:

```
"links": {
  "self": {
    "href": "http://example.com/posts",
  }
}
```

4.7. Jackson annotations

Crnk comes with (partial) support for Jackson annotations. Currently supported are:

Annotation	Description
`@JsonIgnore`	Excludes a given attribute from serialization.
`@JsonProperty.value`	Renames an attribute during serialization.

Annotation	Description
`@JsonProperty.access`	Specifies whether an object can be read and/or written.
`@JsonAnyGetter` and `@JsonAnySetter`	To map dynamic data structures to JSON.

Support for more annotations will be added in the future. PRs welcomed.

5. Repositories

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

- ResourceRepositoryV2 for resources.
- RelationshipRepositoryV2 resp. BulkRelationshipRepositoryV2 for relationships.

5.1. ResourceRepositoryV2

`ResourceRepositoryV2` is the main interface used to operate on resources with `POST`, `GET`, `PATCH` and `DELETE` requests. The interface takes two generic arguments:

1. The type of a resource. Typically this is a plain Java Bean making use of the JSON API annotations. But may also be something entirely different (see [architectures> and <<modules]). One other example is the `io.crnk.core.engine.document.Resource` class used to setup dynamically types repositories.
2. The type of the resource's identifier. Typically a primitive type like `String`, `long` or `UUID`. But if necessary can also be a more complex type that serializes to a URL-friendly String.

The methods of `ResourceRepositoryV2` look 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 that translates into a `404` HTTP error status. The returned resource must adhere to the behavior as specifies by the various annotations (more details in the [resource] chapter), most notably the inclusion of relationships as requested by the passed `querySpec` as long as `LookupIncludeBehavior` does not specify otherwise. More details about `QuerySpec` follow in subsequent sections.
- `findAll(Iterable<ID>ids, QuerySpec querySpec)` Allows to bulk request multiple resources, but otherwise work exactly like the preceding `findOne` method.
- `findAll(QuerySpec querySpec)` Search for all resources according to the passed `querySpec` including sorting, filtering, paging, field sets and inclusions. A `ResourceList` must be returned that carries the result resources, links information and meta information.
- `create(S resource)` Called by `POST` requests. The request body is deserialized and passed as `resource` parameter. The method may or may not have to generate an ID for the newly created resource. The request body may specify relationship data to point to other resources. During deserialization, those resources are looked up and the `@JsonApiRelation` annotated fields set

accordingly. For relationships making use of `@JsonApiRelationId` annotation, only the identifier will be set without the resource annotation, allowing to improve for performance. The `create` method has to save those relationships, but it does and must not perform any changes on the related resources. For bulk inserting and updating resources, have a look at the operations module. The method must return the updated resource, most notably with a valid identifier.

- `save(S resource)` Saves a resource upon a `PATCH` request. The general semantics is identical to the `create(…)` method, with two notable exceptions. First, resources are updated but are not allowed to be inserted. A `ResourceNotFoundException` must be thrown if the resource does not exist. Second, the `PATCH` request allows for partial updates. Internally Crnk will get the current state of a resource, patch it and pass it to this `save` method. API to distinguish patched from un-patched fields is not yet (directly) available, feel free to open a ticket.
- `delete(ID id)` Removes a resource identified by id parameter. A `ResourceNotFoundException` must be thrown if the resource does not exist.

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:

Task.java

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

    @JsonApiId
    private Long id;

    @JsonProperty("name")
    private String name;

    @Size(max = 20, message = "Description may not exceed {max} characters.")
    private String description;

    @JsonApiRelationId
    private Long projectId;

    @JsonApiRelation(opposite = "tasks", lookUp =
LookupIncludeBehavior.AUTOMATICALLY_WHEN_NULL,
            repositoryBehavior = RelationshipRepositoryBehavior.FORWARD_OWNER,
            serialize = SerializeType.ONLY_ID)
    private Project project;

   ...
}
```

and

TaskRepositoryImpl.java

```
@Component
public class TaskRepositoryImpl extends ResourceRepositoryBase<Task, Long> implements
```

```
TaskRepository {

    // for simplicity we make use of static, should not be used in real applications
    private static final Map<Long, Task> tasks = new ConcurrentHashMap<>();

    private static final AtomicLong ID_GENERATOR = new AtomicLong(4);

    public TaskRepositoryImpl() {
        super(Task.class);
    }

    @Override
    public <S extends Task> S save(S entity) {
        if (entity.getId() == null) {
            entity.setId(ID_GENERATOR.getAndIncrement());
        }
        tasks.put(entity.getId(), entity);
        return entity;
    }

    @Override
    public <S extends Task> S create(S entity) {
        if (entity.getId() != null && tasks.containsKey(entity.getId())) {
            throw new BadRequestException("Task already exists");
        }
        return save(entity);
    }

    @Override
    public Class<Task> getResourceClass() {
        return Task.class;
    }

    @Override
    public Task findOne(Long taskId, QuerySpec querySpec) {
        Task task = tasks.get(taskId);
        if (task == null) {
            throw new ResourceNotFoundException("Task not found!");
        }
        return task;
    }

    @Override
    public ResourceList<Task> findAll(QuerySpec querySpec) {
        return querySpec.apply(tasks.values());
    }


    @Override
    public void delete(Long taskId) {
        tasks.remove(taskId);
```

```
    }
}
```

The example is taken from crnk-examples/spring-boot-example. (the basic Spring boot example from crnk-framework, not the dedicated full-blown one from crnk-example).

Together with matching project repository, some URLs to checkout:

```
http://127.0.0.1:8080/api/tasks
http://127.0.0.1:8080/api/tasks/1
http://127.0.0.1:8080/api/tasks/1
http://127.0.0.1:8080/api/tasks/1/project
http://127.0.0.1:8080/api/tasks/1/relationships/project
http://127.0.0.1:8080/api/tasks?sort=-name
http://127.0.0.1:8080/api/tasks?sort=-id,name
http://127.0.0.1:8080/api/tasks?sort=-id,name
http://127.0.0.1:8080/api/tasks?sort=id&page[offset]=0&page[limit]=2
http://127.0.0.1:8080/api/tasks?filter[name]=Do things
http://127.0.0.1:8080/api/tasks?filter[name][EQ]=Do things
http://127.0.0.1:8080/api/tasks?filter[name][LIKE]=Do
http://127.0.0.1:8080/api/tasks?fields=name
http://127.0.0.1:8080/api/projects
http://127.0.0.1:8080/api/tasks?include=project
http://127.0.0.1:8080/api/browse/
```

You may notice that:

- links get automatically created.
- `totalResourceCount` and pagination links are added to the response if the `page` parameter is applied.
- related `project` gets automatically resolved from `projectId`. No relationship repository is implemented here due to the use of `@JsonApiRelationId` (see below).
- the response gets automatically truncated with the `fields` parameter, ideally suited for bandwidth sensitive applications.

This is one small example that **shows the power of native resource-oriented REST libraries**. Implementing a similar API with more classical REST libraries can be a substantial amount of work.

There is further a `ReadOnlyResourceRepositoryBase` base class that does not allow to override the create, delete and update methods. `crnk-meta` accordingly reports insertable, updateable, deltable for such repositories as false.

5.2. RelationshipRepositoryV2

IMPORTANT	Before getting started with the development of relationship repositories, familiarize yourself with @JsonApiRelation.repositoryBehavior. In various scenarios, a custom implementation is unnecessary!

Each relationship defined in Crnk (annotation @JsonApiRelation) must have a relationship repository defined implementing `RelationshipRepositoryV2`. `RelationshipRepositoryV2` implements the methods necessary to work with a relationship. It provides methods for both single-valued and multi-valued relationships:

- `getMatcher()` Provides a `RelationshipMatcher` instance that specifies which relationships it is able provide. It can match against source and target types and fields in any combination. Note that this is a default method that access the legacy `getSourceResourceClass` and `getTargetResourceClass` by default. Implementation of those methods can be omitted if a matcher is available.
- `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 .

All of the methods in this interface have fieldName field as their last parameter in case there are multiple relationships between two resources.

5.2.1. Example

HistoryRelationshipRepository

```
@Component
public class HistoryRelationshipRepository extends
ReadOnlyRelationshipRepositoryBase<Object, Serializable, History, UUID> {

    @Override
    public RelationshipMatcher getMatcher() {
        return new RelationshipMatcher().rule().target(History.class).add();
    }

    @Override
    public ResourceList<History> findManyTargets(Serializable sourceId, String
fieldName, QuerySpec querySpec) {
        DefaultResourceList list = new DefaultResourceList();
        for (int i = 0; i < 10; i++) {
            History history = new History();
            history.setId(UUID.nameUUIDFromBytes(("historyElement" + i).getBytes()));
            history.setName("historyElement" + i);
            list.add(history);
        }
        return list;
    }
}
```

5.2.2. RelationshipMatcher

With `RelationshipRepositoryV2.getMatcher()` one has a lot of flexibility about which kind of relationships a repository is serving. Rules can look like:

RelationshipMatcherTest

```
new RelationshipMatcher().rule().source("projects").add().matches(field)
new RelationshipMatcher().rule().target(Task.class).add().matches(field)
new RelationshipMatcher().rule().target(Tasks.class).add().matches(field)
new RelationshipMatcher().rule().field("tasks").add().matches(field)
new RelationshipMatcher().rule().oppositeField("project").add().matches(field)
[source]
```

One can implement, for example, a history relationship repository that introduces a history relationship for every other resource as done in the example from the previous section.

5.2.3. ForwardingRelationshipRepository

NOTE	Also have a look at @JsonApiRelation.repositoryBehavior before getting started to use this base class.

In many cases, relationship operations can be mapped back to resource repository operations. Making the need for a custom relationship repository implementation redundant.

`@JsonApiRelationId` fields is one example where Crnk will take care of this automatically. But there are many other scenarios where application apply similar techniques. 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 `ForwardingRelationshipRepository` is a base class that takes care of exactly such use cases. `ForwardingRelationshipRepository` knows to `ForwardingDirection`: `OWNER` and `OPPOSITE`. The former forwards requests to the resource repository of the owning side of a relationship, while the later forwards to the opposite side. `ForwardingDirection` is set separately for `GET` and modification operations (`POST`, `PATCH`, `DELETE`).

An example to create such a repository looks like:

```
RelationshipMatcher taskProjectMatcher = new RelationshipMatcher();
taskProjectMatcher.rule().source(Task.class).target(Project.class).add();

new ForwardingRelationshipRepository(
  Task.class, taskProjectMatcher, ForwardingDirection.OWNER, ForwardingDirection.OWNER
);
```

Note that to access the opposite side for `GET` operations, relations must be set up bidirectionally with the `opposite` attribute (to allow filtering in that direction):

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

        @JsonApiRelation(opposite = "tasks", lookUp =
LookupIncludeBehavior.AUTOMATICALLY_WHEN_NULL)
        private Project project;

        ...
    }
```

5.2.4. 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`.

Note that in contrast to `RelationshipRepositoryV2` and `RelationshipRepositoryV2` that are symmetric and applicable to Crnk servers and client, `BulkRelationshipRepositoryV2` is applicable on the server-side only.

5.3. 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.

There is a default implementation named DefaultResourceList. To gain type-safety, improved readability and crnk-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);

    }
```

5.4. Query parameters with QuerySpec

Crnk 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.

NOTE	Not everything is specified by JSON API. For some request parameters only recommendations are provided as different applications are likely to be in need of different semantics and implementations. For this reason the engine part in `crnk-core` makes use of `QueryAdapter` and allows implementations other than QuerySpec (like the legacy `QueryParams`). The mapping of HTTP request parameters to QuerySpec and back is implemented by `QuerySpecUrlMapper`.

For example showing its use URLs also have a look at the ResourceRepositoryV2 section.

5.4.1. Filtering

NOTE	The JSON API specification does not a mandate a specific filtering semantic. Instead it provides a recommendation that comes by default with Crnk. Depending on the data store in use, application may choose to extend or replace that default implementation.

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.

Operators are represented by the `FilterOperator` class. Crnk comes with a set of default filters:

Name	Descriptor
`EQ`	equals operator where values match exactly.
`NEQ`	not equals where values do not match.
`LIKE`	where the value matches the specified pattern. It is usually not case-sensitive and makes use of `%` as wildclard, but may different depending on the underlying implementation.
`LT`	lower than the specified value
`LE`	lower than or equals the specified value
`GT`	greater than the specified value
`GE`	greater than or equals the specified value

The application is free to implements its own `FilterOperator`. Next to the name a `matches` method can be implemented to support in-memory filtering with `QuerySpec.apply`. Otherwise, it is up to the repository implementation to handle the various filter operators; usually by translating them to datastore-native query expressions. Custom operators can be registered with `DefaultQuerySpecUrlMapper.addSupportedOperator(..)`. The default operator can be overridden by setting `DefaultQuerySpecUrlMapper.setDefaultOperator(⋯)`. For more information see `QuerySpecUrlMapper`.

5.4.2. 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`

5.4.3. Pagination

Resources pagination strategies defined by @JsonApiResource.pagingBehavior. By default, each resource has implicit pagination support defined by `OffsetLimitPagingBehavior`. It provides a simple strategy of a pair `offset/limit` parameters which can be used for filtering amount of results.

Pagination executed automatically as soon as client provides a `page` parameter. The format for pagination: `page[offset|limit] = "value", where value is an integer`

Example:

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

The JSON API specifies `first`, `previous`, `next` and `last` links (see http://jsonapi.org/format/#fetching-pagination). The `PagedLinksInformation` interface provides a Java representation of those links that can be implemented and returned by repositories along with the result data. There is a default implementation named `DefaultPagedLinksInformation`.

There are two ways to let Crnk compute pagination links automatically:

1. The repository returns meta information implementing `PagedMetaInformation`. With this interface the total number of (potentially filtered) resources is passed to Crnk, which in turn allows the computation of the links.
2. The repository returns meta information implementing `HasMoreResourcesMetaInformation`. This interface only specifies whether further resources are available after the currently requested resources. This lets Crnk compute all except the `last` link.

Note that for both approaches the repository has to return either no links or links information implementing `PagedLinksInformation`. If the links are already set, then the computation will be skipped.

The potential benefit of the second over the first approach is that it might be easier to just determine whether more resources are available rather than counting all resources. This is typically achieved by querying `limit + 1` resources.

Custom strategies

Resources might have custom pagination strategies. In order to do so, you would have to perform the following actions:

1. Provide an instance of a custom `PagingBehavior` implementation.
2. Set it up via @JsonApiResource.pagingBehavior attribute.

5.4.4. 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`

5.4.5. 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)

5.4.6. API

The QuerySpec 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 PagingSpec getPagingSpec() {...}

    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 a `QuerySpecUrlMapper` implementation. The type is determined based on the type of the filtered attribute.

`PagingSpec` holds a value of defined pagination strategy. By default, the values can be accessed via `getLimit/getOffset` methods. In case of custom strategies, `getPagingSpec` should be used.

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 Crnk automatically compute pagination links.

5.4.7. URL Mapping

The mapping of request parameters to QuerySpec and back is implemented by `QuerySpecUrlMapper`. With `DefaultQuerySpecUrlMapper` there is a default implementation that follows the JSON API specification and recommendations and introduces some further defaults as documented in the previous sections (like the filter operators) where the recommendations do not go far enough. The used `QuerySpecUrlMapper` can be obtained from `CrnkBoot.getUrlMapper()` and

`CrnkClient.getUrlMapper()`. Matching setter allow to setup a custom implementation.

`DefaultQuerySpecUrlMapper` comes with a number of customization options:

- `setAllowUnknownAttributes(boolean)` `DefaultQuerySpecUrlMapper` 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 this should be necessary.
- `setAllowUnknownParameters(boolean)` `DefaultQuerySpecUrlMapper` validates all passed parameters to be one of the following types: `filter`, `sort`, `page`, `fields` or `include`. In case of any custom query parameter `ParametersDeserializationException` will be thrown. This flag allows to disable that check and ignore any unknown ones.
- `setIgnoreParseExceptions(boolean)` `DefaultQuerySpecUrlMapper` attempts to convert all filter parameters to their proper type based on the attribute type to be filtered. In some scenarios like dates this behavior may be undesirable as applications introduce expressions like 'now`. Enabling this flag will let `DefaultQuerySpecDeserializer` ignore such values and provide them as `String` within `FilterSpec`.
- `setEnforceDotPathSeparator(boolean)` DefaultQuerySpecUrlMapper by default supports by default two URL conventions: `task[project][name]=myProject` and `task[project.name]=myProject`. The later is recommended. The former still supported for historic reasons. By default the flag is still disabled, but it is recommended to be enabled and will become the default at some point in the future. Note that without the enforcement, there is danger of introducing ambiguity with resources and attributes are named equally.
- `setMapJsonNames` Whether to map JSON to Java names for `QuerySpec`. Enabled by default. Typically JSON and Java names are equal, but, for example, fields can be renamed with the `@JsonProperty` annotation.
- `addSupportedOperator` Adds a new `FilterOperator`. See [filtering] for more information.
- `setDefaultOperator` Sets the default `FilterOperator`. See [filtering] for more information.

Some of those methods are also available from some of the integrations like `CrnkFeature` for convenience.

5.5. Error Handling

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

5.5.1. 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;
  }
}
```

5.5.2. ...and mapping it to JSON API response

Class responsible for mapping the exception should:

- implement ExceptionMapper interface
- available trough the used discovery mechanism or added trough a module.

Sample exception mapper:

TestExceptionMapper.java

```
package io.crnk.test.mock;

import java.util.List;

import io.crnk.core.engine.document.ErrorData;
import io.crnk.core.engine.error.ErrorResponse;
import io.crnk.core.engine.error.ExceptionMapper;
import io.crnk.core.repository.response.JsonApiResponse;

public class TestExceptionMapper implements ExceptionMapper<TestException> {

    public static final int HTTP_ERROR_CODE = 499;

    @Override
    public ErrorResponse toErrorResponse(TestException cve) {
        ErrorData error = ErrorData.builder().setDetail(cve.getMessage()).build();
        return
ErrorResponse.builder().setStatus(HTTP_ERROR_CODE).setSingleErrorData(error).build();
    }

    @Override
    public TestException fromErrorResponse(ErrorResponse errorResponse) {
        JsonApiResponse response = errorResponse.getResponse();
        List<ErrorData> errors = (List<ErrorData>) response.getEntity();
        StringBuilder message = new StringBuilder();
        for (ErrorData error : errors) {
            String title = error.getDetail();
            message.append(title);
        }
        return new TestException(message.toString());
    }

    @Override
    public boolean accepts(ErrorResponse errorResponse) {
        return errorResponse.getHttpStatus() == HTTP_ERROR_CODE;
    }
}
```

On the server-side an exception should be mapped to an `ErrorResponse` object with `toErrorResponse`. It consists of an HTTP status and `ErrorData` (which is consistent with JSON API error structure). On the client-side an `ExceptionMapper` returning `true` upon `accept(…)` is used to map an `ErrorResponse` back to an exception with `fromErrorResponse`.

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.

5.6. 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.crnk.core.repository.MetaRepository`. It contains a single method `MetaInformation getMetaInformation(Iterable<T> resources)` which return meta information object that implements the marker `interface io.crnk.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.

5.7. Links Information

NOTE	With ResourceList and @JsonApiLinksInformation links information can be returned directly. A LinksRepository implementation is usually not necessary.

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

5.8. Payload Size Optimizations

Self and related links can make up to 60% of the response payload size and not always are those links of use. Crnk offers a `Crnk-Compact: true` header that can be sent along with the request. In this case the computation of those links is omitted. Further relationships without data are completely omitted.

5.9. Repository Decoration

Sometimes it is useful to augment a repository with further features. There are different situations where that can makes sense:

- the main repository implementation comes from a third-party library and can/should not directly be modified.
- the feature is unrelated to the main repository feature set, for example, cross-cutting concerns like security, caching, tracing and metrics.

`RepositoryDecoratorFactory` allows to do exactly that. It puts a **decorating** resource or relationship repository inbetween the Crnk engine and the main repository. It implements the same interface contract as the main repository, intercept all requests and can do arbitrary modifications. The modifications may or may not include calling the main repository.

An example can look like:

ApprovalRepositoryDecorator.java

```
    public static final RepositoryDecoratorFactory createFactory(ApprovalManager
approvalManager) {
        return new RepositoryDecoratorFactoryBase() {

            @Override
            public <T, I extends Serializable> ResourceRepositoryDecorator<T, I>
decorateRepository(
                    ResourceRepositoryV2<T, I> repository) {
                if (repository.getResourceClass() == Schedule.class) {
                    return (ResourceRepositoryDecorator<T, I>) new
ApprovalRepositoryDecorator(approvalManager);
                }
                return null;
            }
        };
    }
```

The particular example, for example, intercepts the `save` operation and may trigger an approval workflow:

ApprovalRepositoryDecorator.java

```
    @Override
    public <S extends T> S save(S entity) {
        if (approvalManager.needsApproval(entity, HttpMethod.PATCH)) {
            return approvalManager.requestApproval(entity, HttpMethod.PATCH);
        }
        else {
            return super.save(entity);
        }
    }
```

5.10. ResourceFieldContributor

The `ResourceFieldContributor` interface allows to dynamically introduce new fields to resources without actually touching them. This is useful, for example, if you have a JPA entity exposed with crnk-jpa and want to add further fields like that mentioned history relationship from the earlier relationship example. `ResourceFieldContributor` can be implemented by a repository or obtained from the regular service discovery mechanism.

Any type of field can be added: meta, links, attribute and relationship fields. For relationship fields an application may make use of `RelationshipMatcher` to provide repository serving those fields. Notice the `accessor` property that is used to obtain the value of that field (make sure this method is efficient to execute). An example is given by the `HistoryRelationshipRepository` in crnk-examples/spring-boot-example:

HistoryFieldContributor

```
@Component
public class HistoryFieldContributor implements ResourceFieldContributor {

    @Override
    public List<ResourceField> getResourceFields(ResourceFieldContributorContext
context) {
        // this method could be omitted if the history field is added regularly to
Project and Task resource. This would be
        // simpler and recommended, but may not always be possible. Here we
demonstrate doing it dynamically.
        InformationBuilder.Field fieldBuilder =
context.getInformationBuilder().createResourceField();
        fieldBuilder.name("history");
        fieldBuilder.genericType(new TypeToken<List<History>>() {
        }.getType());
        fieldBuilder.oppositeResourceType("history");
        fieldBuilder.fieldType(ResourceFieldType.RELATIONSHIP);

        // field values are "null" on resource and we make use of automated lookup to
the relationship repository
        // instead:

fieldBuilder.lookupIncludeBehavior(LookupIncludeBehavior.AUTOMATICALLY_ALWAYS);
        fieldBuilder.accessor(new ResourceFieldAccessor() {
            @Override
            public Object getValue(Object resource) {
                return null;
            }

            @Override
            public void setValue(Object resource, Object fieldValue) {
            }
        });
        return Arrays.asList(fieldBuilder.build());
    }
}
```

- `getRelationshipFields` introduces the field dynamically instead of statically.
- `getMatcher` attaches the repository to the historized resources.
- `findManyTarget` implements the lookup of history elements. :basedir: ../../../.. :clientdir: ../../../../crnk-client

6. Client

There is a client implementation for Java and Android projects to allow communicating with JSON-API compliant servers.

6.1. Setup

The basic setup is as simple as:

```
CrnkClient client = new CrnkClient("http://localhost:8080/api");
```

Three underlying http client libraries are supported:

- OkHttp that is popular for Java and Android development. Implemented by `io.crnk.client.http.okhttp.OkHttpAdapter`.
- Apache Http Client implemented by `io.crnk.client.http.apache.HttpClientAdapter`.
- `RestTemplate` from Spring provides a Spring abstraction of other HTTP libraries. Spring application benefit from using this over the underlying native implementation to share the setup configuration setup. It is used by default if the presence of Spring is detected. Implemented by `io.crnk.spring.client.RestTemplateAdapter`.

Add one of those library to the classpath and Crnk will pick it up automatically. A custom `HttpAdapter` can also set passed to `CrnkClient.setHttpAdapter(…)`.

WARNING	For Spring a **reasonable HTTP client implementation must underlie `RestTemplate`** in order for `crnk-client` to work properly. For example, the default Java implementation does not support the `PATCH` method and as such no resources can be updated. To explicitly set HTTP implementation use:

```
RestTemplateAdapter adapter = (RestTemplateAdapter) client.getHttpAdapter();
RestTemplate template = adapter.getImplementation();
template.setRequestFactory(new OkHttp3ClientHttpRequestFactory());
```

or

```
client.setHttpAdapter(new RestTemplateAdapter(customRestTemplate));
```

6.2. Usage

The client has three main methods:

- `CrnkClient#getRepositoryForInterface(Class)` to obtain a resource repository stub from an existing repository interface.

- `CrnkClient#getRepositoryForType(Class)` to obtain a generic resource repository stub from the provided resource type.
- `CrnkClient#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:

```
CrnkClient client = new CrnkClient("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.

6.3. URL handling

Crnk clienet and server share the URL handling. `QuerySpecUrlMapper` performs the mapping of HTTP request parameters to QuerySpec. For more information see url mapping.

6.4. Modules

`CrnkClient` can be extended by modules:

```
CrnkClient client = new CrnkClient("http://localhost:8080/api");
client.addModule(ValidationModule.create());
```

Typical use cases include:

- adding exception mappers
- registering new types of resources (like JPA entities by the `JpaModule`)
- intercepting requests for monitoring
- adding security tokens to requests

Many modules allow a registration both on server and client side. The client part then typically makes use of a subset of the server features, like exception mappers and resource registrations.

There is a mechanism to discover and register client modules automatically:

```
CrnkClient client = new CrnkClient("http://localhost:8080/api");
client.findModules();
```

`findModules` makes use of `java.util.ServiceLoader` and looks up for `ClientModuleFactory`. `JpaModule`, `ValidationModule`, `MetaModule`, `SecurityModule` implement such a service registration. In contrast,

`BraveModule` needs a Brave instance and does not yet allow a fully automated setup.

6.5. Type-Safety

It is possible to work with `CrnkClient` in a fully type-safe manner.

In a first step an interface for a repository is defined:

ScheduleRepository.java

```
public interface ScheduleRepository extends ResourceRepositoryV2<Schedule, Long> {
    @Override
    ScheduleList findAll(QuerySpec querySpec);

    class ScheduleList extends ResourceListBase<Schedule, ScheduleListMeta,
ScheduleListLinks> {

    }

    class ScheduleListLinks extends DefaultPagedLinksInformation implements
LinksInformation {

        public String name = "value";
    }

    class ScheduleListMeta implements MetaInformation {

        public String name = "value";

    }
}
```

And then it can be used like:

QuerySpecClientTest.java

```
        ScheduleRepository scheduleRepository = ((ClientTestContainer)
testContainer).getClient().getResourceRepository(ScheduleRepository.class);

        Schedule schedule = new Schedule();
        schedule.setId(13L);
        schedule.setName("mySchedule");
        scheduleRepository.create(schedule);

        QuerySpec querySpec = new QuerySpec(Schedule.class);
        ScheduleList list = scheduleRepository.findAll(querySpec);
        Assert.assertEquals(1, list.size());
        ScheduleListMeta meta = list.getMeta();
        ScheduleListLinks links = list.getLinks();
        Assert.assertNotNull(meta);
        Assert.assertNotNull(links);
```

6.6. JAX-RS interoperability

The interface stubs from the previous section can also be used to make calls to JAX-RS. For example, the `ScheduleRepository` can be complemented with a JAX-RS annotation:

ScheduleRepository.java

```
@Path("schedules")
@Produces(HttpHeaders.JSONAPI_CONTENT_TYPE)
```

and further JAX-RS services can be added:

ScheduleRepository.java

```
    @GET
    @Path("repositoryAction")
    @Produces(MediaType.TEXT_HTML)
    String repositoryAction(@QueryParam(value = "msg") String msg);

    @GET
    @Path("repositoryActionJsonApi")
    String repositoryActionJsonApi(@QueryParam(value = "msg") String msg);

    @GET
    @Path("repositoryActionWithJsonApiResponse")
    String repositoryActionWithJsonApiResponse(@QueryParam(value = "msg") String msg);

    @GET
    @Path("repositoryActionWithResourceResult")
    Schedule repositoryActionWithResourceResult(@QueryParam(value = "msg") String
msg);

    @GET
    @Path("repositoryActionWithException")
    Schedule repositoryActionWithException(@QueryParam(value = "msg") String msg);

    @GET
    @Path("repositoryActionWithNullResponse")
    @Produces(MediaType.TEXT_HTML)
    String repositoryActionWithNullResponse();

    @GET
    @Path("repositoryActionWithNullResponseJsonApi")
    String repositoryActionWithNullResponseJsonApi();

    @GET
    @Path("{id}/resourceAction")
    String resourceAction(@PathParam("id") long id, @QueryParam(value = "msg") String
msg);
```

To make this work a dependency to `org.glassfish.jersey.ext:jersey-proxy-client` must be added and `JerseyActionStubFactory` registered with `CrnkClient`:

AbstractClientTest.java

```
        client.setActionStubFactory(JerseyActionStubFactory.newInstance());
```

Then a client can make use the Crnk stubs and it will transparently switch between JSON-API and JAX-RS calls:

JsonApiActionResponseTest.java

```
        String result = scheduleRepository.repositoryAction("hello");
        Assert.assertEquals("repository action: hello", result);
```

WARNING	Due to limited configurability of the Jersey Proxies it is currently not possible to reuse the same HTTP connections for both types of calls. We attempt to address that in the future. Be aware of this when you, for example, add further request headers (like security), as it has to be done in two places (unfortunately).

6.7. HTTP customization

It is possible to hook into the HTTP implementation used by Crnk (or Apache). Make use of `CrnkClient#getHttpAdapter()` and cast it to either `HttpClientAdapter` or `Adapter`. 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 `crnk-brave` for an advanced example.

7. Reactive Programming

WARNING	Initial support is available, but still considered (very) experimental. The implementation is expected to mature over the coming weeks. Breaking changes to the reactive API might be possible. The traditional API is left unchanged.

The `ReactiveModule` of `crnk-reactive` bring support for reactive programming to Crnk. It allows to build more responsive, elastic, resilient, message-driven applications (see https://www.reactivemanifesto.org/). https://projectreactor.io/ was chosen as library.

IMPORTANT	Traditional and reactive-style programming APIs are considered being equally important and will both be supported the coming years.

`crnk-reactive` brings along three new interfaces that act as reactive counter-parts of the traditional resource and relationship interfaces:

- `ReactiveResourceRepository`
- `ReactiveOneRelationshipRepository`
- `ReactiveManyRelationshipRepository`

The differences to the traditional ones are:

- Single and multi-valued relationships are served by different interfaces (minor cleanup, usually one or the other is necessary).
- `ResourceField` instead of a simple `String` give more detailed information about the accessed relationship.
- Most importantly, `reactor.core.publisher.Mono` is used as return type to enable reactive programming.

NOTE that:

- A potential future V3 version of the traditional interfaces will align the first two differences.
- `Mono` rather than `Flux` is used for list return types since meta and links information must be returned as well, not just a sequence of resources. For large number of resources, the JSON API pagination mechanisms can be applied.
- Internally the traditional and reactive repositories are served by the same Crnk engine and share virtually all of the code base. The difference lies in the used `ResultFactory` implementation. `ImmediateResultFactory` is used by the traditional API. `MonoResultFactory` by reactive setups.

7.1. Servlet Example

The subsequent example shows as simple reactive resource repository holding its data in-memory:

InMemoryReactiveResourceRepository.java

```
package io.crnk.test.mock.reactive;

import io.crnk.core.engine.information.resource.ResourceField;
import io.crnk.core.engine.internal.utils.PreconditionUtil;
import io.crnk.core.engine.registry.RegistryEntry;
import io.crnk.core.queryspec.QuerySpec;
import io.crnk.core.resource.list.ResourceList;
import io.crnk.reactive.repository.ReactiveResourceRepositoryBase;
import io.crnk.test.mock.TestException;
import io.crnk.test.mock.UnknownException;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import reactor.core.publisher.Mono;

import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

public class InMemoryReactiveResourceRepository<T, I> extends
ReactiveResourceRepositoryBase<T, I> {

    private static final Logger LOGGER =
LoggerFactory.getLogger(InMemoryReactiveResourceRepository.class);
```

```
    protected Map<I, T> resources = new ConcurrentHashMap<>();

    private long nextId = 0;

    public InMemoryReactiveResourceRepository(Class<T> clazz) {
        super(clazz);
    }


    @Override
    public Mono<T> findOne(I id, QuerySpec querySpec) {
        if ((Long) id == 10000L) {
            return Mono.error(new TestException("msg"));
        }
        if ((Long) id == 10001L) {
            return Mono.error(new UnknownException("msg"));
        }
        return super.findOne(id, querySpec);
    }


    @Override
    public Mono<ResourceList<T>> findAll(QuerySpec querySpec) {
        LOGGER.debug("findAll {}", querySpec);
        return Mono.fromCallable(() -> querySpec.apply(resources.values()));
    }

    @Override
    public Mono<T> create(T entity) {
        LOGGER.debug("create {}", entity);

        RegistryEntry entry = resourceRegistry.findEntry(getResourceClass());
        ResourceField idField = entry.getResourceInformation().getIdField();
        Long id = (Long) idField.getAccessor().getValue(entity);
        if (id == null) {
            idField.getAccessor().setValue(entity, nextId++);
        }
        return save(entity);
    }


    @Override
    public Mono<T> save(T entity) {
        LOGGER.debug("save {}", entity);
        RegistryEntry entry = resourceRegistry.findEntry(getResourceClass());
        ResourceField idField = entry.getResourceInformation().getIdField();
        I id = (I) idField.getAccessor().getValue(entity);
        PreconditionUtil.assertNotNull("no id specified", entity);
        if ((Long) id == 10000L) {
            return Mono.error(new TestException("msg"));
```

```
        }
        if ((Long) id == 10001L) {
            return Mono.error(new UnknownException("msg"));
        }
        resources.put(id, entity);
        return Mono.just(entity);
    }

    @Override
    public Mono<Boolean> delete(I id) {
        LOGGER.debug("delete {}", id);
        return Mono.fromCallable(() -> resources.remove(id) != null);
    }

    public Map<I, T> getMap() {
        return resources;
    }

    public void clear() {
        resources.clear();
    }
}
```

The following snipped shows how to setup the `ReactiveModule` together with `AsyncCrnkServlet` in Spring:

ReactiveServletTestApplication.java

```
    @Bean
    public AsyncCrnkServlet asyncCrnkServlet(SlowResourceRepository
slowResourceRepository) {
        SimpleModule slowModule = new SimpleModule("slow");
        slowModule.addRepository(slowResourceRepository);

        AsyncCrnkServlet servlet = new AsyncCrnkServlet();
        servlet.getBoot().addModule(new ReactiveModule());
        servlet.getBoot().addModule(testModule);
        servlet.getBoot().addModule(slowModule);

        return servlet;
    }

    @Bean
    public ServletRegistrationBean crnkServletRegistration(AsyncCrnkServlet servlet) {
        ServletRegistrationBean bean = new ServletRegistrationBean(servlet, "/api/*");
        bean.setLoadOnStartup(1);
        return bean;
    }

    @Bean
    public CrnkBoot crnkBoot(AsyncCrnkServlet servlet) {
        return servlet.getBoot();
    }
```

Ready-to-use integrations into Spring, Vert.x and Ratpack are planned for the near future.

7.2. Interacting with traditional repositories

Reactive and traditional repositories work well together side-by-side and can also be used in a mixed setting, for example when requesting the inclusion of related resources. Since the traditional repositories will block the caller, there are two mechanisms in place to remedy the issue:

1. A repository can be annotated with `ImmediateRepository`. In this case the traditional repository is marked as being non-blocking and can be safely invoked.
2. `ReactiveModule.setWorkerScheduler(⋯)` allows to set the scheduler to use to invoke traditional repositories. By default `Schedulers.elastic` is used where worker threads are spanned as necessary.

7.3. Roadmap and Limitations

- `DocumentFilter`, `RepositoryFilter` cannot be used in a reactive setup. Reactive counter-party will be available in the near future.
- `HttpRequestContextProvider.getRequestContext` can only be used in a non-reactive setting or while the reactive request is being setup. `HttpRequestContextProvider.getRequestContextResult`

must be used instead that makes use of the subscriber context of Reactor.

- `crnk-jpa` has not yet been ported yet. JDBC and JPA are blocking and require a transaction.
- Spring, Vert.x and Ratpack integrations are target for the near future.
- More testing will be necessary.

Contributions in any area welcomed. :basedir: ../../../..

8. Security

The resource-oriented nature of Crnk makes a number of quite powerful security schemes possible to authorize access to resources. The semantics of JSON API with resources, relationships, fields and parameters provide much more information about an application compared to, for example, a more classical JAX-RS oder Spring MVC application. This in turn allows to automate and simplify many security-related topics, which in turn allows for a more robust authorization.

Authorization can happen on three levels:

- Resource-based authorization: only a subset of all resource types are accessible to the user.
- Field-based authorization: only a subset of the fields (attributes and relationships) of a resource are accessible to the user.
- Data-based authorization: only a subset of all resources of a given type are accessible to the user based on the contents of the resources. Typically denominated Dataroom access control (DAC).

Crnk comes with support for all three. The subsequent sections outline a number of different strategies how to apply them.

8.1. Authentication

Authorization requires to first know the calling user through authentication. Crnk is agnostic to the used authorization scheme and is usually provided by the underlying integration, for example, the `Principal` of JEE or the `SecurityContext` of Spring Security. They in turn are populated from a scheme like OAuth, JWT or SAML.

Crnk makes use of `SecurityProvider` to integrate with such systems and check access to roles for the current user:

SecurityProvider.java

```
package io.crnk.core.engine.security;

public interface SecurityProvider {

    boolean isUserInRole(String role);
}
```

The JAX-RS and servlet integration of Crnk come both with a `SecurityProvider` implementation. Which in turn also provides an implementation for all integrations derived from them, such as Spring.

The `SecurityProvider` is accessible from the module API and can be used to perform both DAC and RBAC or any other kind of check.

8.2. Resource-based Access Control with the SecurityModule

There is a `SecurityModule` provided by `crnk-security` that intercepts all repository requests and perform access control. Currently it supports resource-based access control. A setup can looks as follows:

SecurityModuleIntTest.java

```
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);
module = SecurityModule.newServerModule(builder.build());

CrnkFeature feature = new CrnkFeature();
feature.addModule(module);
```

A builder is used to construct rules. Each rule grants access to either a given or all resources. Thereby `ResourcePermission` specifies the set of authorized methods: `GET`, `POST`, `PATCH`, `DELETE`.

Once the rules are defined, the runtime checks go well beyond more traditional approaches like JEE `@RolesAllowed` annotation. The rules are enforced in various contexts:

- Access to resource repositories are checked.
- Access to relationship repositories are checked based on the target (return) type.
- Relationship fields targeting resources the user is not authorized to see are omitted from results and cannot be modified.
- A request may span multiple repository accesses in case of inclusions with the `include` parameter. In this case every access is checked individually.
- `HomeModule` and `MetaModule` show only resources the user is authorized to see. In case of the `MetaModule` the `MetaAttribute` and `MetaResource` provide further information about what can be read, inserted, updated and deleted.

- (soon) Query parameters for sorting and filtering are checked against unauthorized access to related resources.

Internally the security module makes use of `ResourceFilter` to perform this task. More information about that is available in a subsequent section.

Is is up to the application how and when to configure the `SecurityModule`. The set of rules can be static or created dynamically. `SecurityModule.reconfigure(⋯)` allows to replace the security rules at runtime.

8.3. Role-based Access Control with ResourceFilter

The `SecurityModule` is only one example how to implement security. Underlying it is the `ResourceFilter` interface provided by the Crnk engine:

ResourceFilter.java

```
public interface ResourceFilter {
    FilterBehavior filterResource(ResourceInformation resourceInformation, HttpMethod
method);
    FilterBehavior filterField(ResourceField field, HttpMethod method);
}
```

`ResourceFilter` allows to restrict access to resources and fields. To methods `filterResource` and `filterField` can be implemented for this purpose. Both return a `FilterBehavior` which allows to distinguish between `NONE`, `IGNORE` and `FORBIDDEN`. For example, a field like a lock count can make use of `IGNORE` in order to be ignored for POST and PATCH requests (the current value on the server is left untouched). While access to an unauthorized resource or field results in a forbidden error with `FORBIDDEN`. An example is given by the `SecurityResourceFilter` of `SecurityModule` in 'crnk-security`. Since `ResourceFilter` methods are invoked often, it is important for them to return quickly.

There is a `ResourceFilterDirectory` that complements `ResourceFilter`. It allows to query the authorization status of a particular resource or field in context of the current request. The `ResourceFilterDirectory` makes use of per-request caching as the information may be accessed repeatedly for a request. It can be obtained with `ModuleContext.getResourceFilterDirectory(⋯)` from the module API. For example, the `MetaModule` and `HomeModule` make use of `ResourceFilterDirectory` to only list authorized elements.

8.4. Dataroom Access Control

Filtering of resources is one of the main building blocks of JSON API. As such it is typically not too hard to implement DAC. The roles of a user can be checked and if necessary for filters specific to that user can be added. There are two possibilities to add such filters:

- by updating the repository filters.
- by implementing a repository decorator that intercepts the request before reaching the

repository. For more information see Repository Decoration.

In the future the `SecurityModule` may be enhanced to also support DAC.

8.5. Adapt User Interfaces based on Authorizations

In many cases it is desired to adjust UIs based on the authorizations of a user to guide the user early what he is authorized to do. There are two mechanisms that are outlined in the next sections.

8.5.1. ResourcePermissionInformation

The `ResourcePermissionInformation` interface specializes `MetaInformation` to give access to the `ResourcePermission` of that particular element, either a list or a single resource. If either of the two carries a `MetaInformation` implementing `ResourcePermissionInformation`, then the `SecurityModule` will fill-in the `ResourcePermission` for the current request.

8.5.2. Home and Meta Module

- `HomeModule` and `MetaModule` hide resources the user is not authorized to see. Any UI can query their respective URLs to gain information about what the user is authorized to see.
- The `MetaAttribute` and `MetaResource` resources from the `MetaModule` further show information about which resources and fields can be inserted, updated and deleted. The resources are available from `/meta/attributes` and `/meta/resource` respectively.

8.6. Provide Authentication Information to the User

A frequent use case is to display user/login related information for a user. However, Crnk does not do authentication on its own and the set of provided information is typically application-specific. As such there is no direct support from Crnk, but a custom repository can look like:

LoginRepository.java

```
public class LoginRepository extends ResourceRepositoryBase<Login, String> {


    public LoginRepository() {
        super(Login.class);
    }

    @Override
    public ResourceList<Login> findAll(QuerySpec querySpec) {
        List<Login> logins = new ArrayList<>();
        SecurityContext context = SecurityContextHolder.getContext();
        if (context != null) {
            Authentication authentication = context.getAuthentication();
            Login me = new Login();
            me.setId("me");
            me.setUserName(authentication.getName());
            logins.add(me);
        }
        return querySpec.apply(logins);
    }
}
```

This particular example has been taken from crnk-example.

8.7. Exception Mapping

Crnk comes with four exceptions that are relevant in a security context:

- `io.crnk.core.exception.ForbiddenException` results in a HTTP `403` status code and forbids access to the requested element.
- `io.crnk.core.exception.UnauthorizedException` results in a HTTP `401` status code to trigger authentication.
- `io.crnk.core.exception.RepositoryNotFoundException` and `io.crnk.core.exception.ResourceNotFoundException` may be used in favor of `io.crnk.core.exception.ForbiddenException` to completely hide unauthorized resources with a status `404` indistinguishable from non-existing ones.

8.8. (Potential) Future Work

- Authorize access to fields with `SecurityModule`.
- Authorize sort and filter parameters.
- Resource annotations to configure `SecurityModule`.
- Deny rules for `SecurityModule`.
- Potentially give security-related information through `OPTIONS` requests.

- DAC support for `SecurityModule`.

9. Modules

9.1. JPA Module

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

The feature set includes:

- expose JPA entities to JSON API repositories
- expose JPA relations as JSON API repositories
- decide which entities to expose as endpoints
- sorting, filtering, paging, inclusion of related resources.
- all default operators of crnk are supported: `EQ`, `NEQ`, `LIKE`, `LT`, `LE`, `GT`, `GE`.
- filter, sort and include parameters can make use of the dot notation to join to related entities. For example, `sort=-project.name,project.id`, `filter[project.name][NEQ]=someValue` or `include=project.tasks`.
- support for entity inheritance by allowing sorting, filtering and inclusions to refer to attributes on subtypes.
- support for Jackson annotations to customize entity attributes on the JSON API layer, see here.
- DTO mapping support to map entities to DTOs before sending them to clients.
- JPA Criteria API and QueryDSL support to issue queries.
- filter API to intercept and modify issued queries.
- support for computed attributes behaving like regular, persisted attributes.
- automatic transaction handling spanning requests and doing a rollback in case of an exception.
- `OptimisticLockExceptionMapper` mapped to JSON API errors with `409` status code.
- `PersistenceException` and `RollbackException` are unwrapped to the usually more interesting exceptions like `ValidationException` and then translated to JSON API errors.

Have a look at the Spring Boot example application which makes use of the JPA module, DTO mapping and computed attributes.

Not yet supported are sparse field sets queried by tuple queries.

9.1.1. JPA Module Setup

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

```
    TransactionRunner transactionRunner = ...;

    JpaModuleConfig config = new JpaModuleConfig();

    // expose all entitities from provided EntityManagerFactory
    config.exposeAllEntities(entityManagerFactory);

    // expose single entity
    config.addRepository(JpaRepositoryConfig.builder(TaskEntity.class).build());

    // expose single entity with additional interface declaration. Interface is used
to
    // extract the list, meta and link information types.
    config.addRepository(
      JpaRepositoryConfig.builder(PersonEntity.class)
           .setInterfaceClass(PersonRepository.class).build()
    );

    JpaModule jpaModule = JpaModule.createServerModule(config, em,
transactionRunner());

    CrnkFeature feature = new CrnkFeature(...);
    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 Crnk using the module API. Accordingly, every relation is registered as `JpaRelationshipRepository`.

- `JpaModuleConfig.setRepositoryFactory` allows to provide a factory to change or customized the used repositories.
- `exposeAllEntities` takes an `EntityManagerFactory` and exposes all registered entities as JSON API repository.
- To manually select the entities exposed to Crnk use `JpaModuleConfig.addRepository(⋯)`. `JpaRepositoryConfig` provides a number of customization options for the exposed entity. `setListClass`, `setListMetaClass` and `setListLinksClass` allow to set the type information and will then return resources having lists, links and meta data of the given type. `setInterfaceClass` is a shortcut that allows to extract those three types from a repository interface definition (see `TaskRepository` below for an example).
- `JpaRepositoryConfig.Builder.setRepositoryDecorator` allows to setup a repository decorator that allows to intercept the request and responses and perform changes, like setting up additional links and meta information.
- `JpaRepositoryConfig` allows to specify DTO mapping, have a look at the later section to this topic.
- Internally the JpaModule will create instances of `JpaResourceRepository` and `JpaRelationshipRepository` to register them
- 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. Crnk comes with two default implementations: `SpringTransactionRunner` and `CdiTransactionRunner` that come are included in `crnk-setup-spring` and `crnk-cdi`.

For crnk-client it might be worthwhile to have an interface definition of the otherwise fully generic JSON API entity repositories. The interface has the benefit of providing proper typing of meta information, link information and list return type. An example can look like:

```
public interface TaskRepository extends ResourceRepositoryV2<TaskEntity, UUID> {

    static class TaskListLinks implements LinksInformation, SelfLinksInformation {

        public String someLink = "test";

    }

    public static class TaskListMeta implements MetaInformation {

        public String someMeta = "test";

    }

    public class TaskList extends ResourceListBase<TaskEntity, TaskListMeta,
TaskListLinks> {

    }

    @Override
    public TaskList findAll(QuerySpec querySpec);
}
```

To setup a Crnk client with the JPA module use:

```
    client = new CrnkClient(getBaseUri().toString());

    JpaModule module = JpaModule.newClientModule();
    setupModule(module, false);
    client.addModule(module);

    ResourceRepositoryV2<TaskEntity, UUID> genericRepo =
client.getRepositoryForType(TypeEntity.class)
    TaskRepository typedRepo = client.getRepositoryForInterface(TaskRepository.class)
```

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

9.1.2. JPA Entity Setup

Most features of JPA are supported and get mapped to JSON API:

- Entities are mapped to resources.
- Embeddables are mapped to nested json structures.
- Embeddables used as primary keys are mapped to/from a simple string to remain addressable as resource id. The order of attributes thereby determines the position of the values in the string.
- Not yet supported are relationships within embeddables.

It is possible to add additional JSON API related fields to entities by making use of the `@Transient` annotation of `javax.persistence` (or the other way around by marking it with `@JsonIgnore`):

JpaTransientTestEntity.java

```
@Entity
public class JpaTransientTestEntity extends TestMappedSuperclass {

    @Id
    private Long id;

    @Transient
    @JsonApiRelation(serialize = SerializeType.LAZY, lookUp =
LookupIncludeBehavior.NONE)
    private Task task;

    public Long getId() {
        return id;
    }

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

    public Task getTask() {
        return task;
    }

    public void setTask(Task task) {
        this.task = task;
    }
}
```

However, for more complex scenarios application may rather make use of DTO mapping.

9.1.3. JPA Setup with Spring

Have a look at JPA Module Auto Configuration. For Spring Boot an auto configuration is available

that already takes care of some defaults.

9.1.4. Pagination

The JPA module implements both pagination approaches supported by Crnk. Setting `JpaModule.setTotalResourceCountUsed(true|false)` allows to decide whether the total number of resources should be counted or whether just the presence of a subsequent resource is checked (by querying `limit + 1` entities). By default the total resources are counted. Have a look at the [pagination] section for more information.

9.1.5. 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.

9.1.6. Customizing the JPA repository

In general JPA repositories support the same set of decoration and filtering mechanisms as any other repository. They allow to intercept and customize request before or after hitting the JPA repositories. Next to that, there are further mechanisms to customize the JPA repositories themselves. 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.

9.1.7. 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.addRepository(JpaRepositoryConfig.builder(TestEntity.class, TestDTO.class,
new TestDTOMapper(entityManager)).build())
```

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/crnk-project/crnk-framework/blob/master/crnk-jpa/src/test/java/io/crnk/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.

9.2. JSR 303 Validation Module

A `ValidationModule` provided by `io.crnk:crnk-validation` implements resource validation and provides 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.

A translated exception can look like:

```
{
    "errors": [
        {
            "status": "422",
            "code": "javax.validation.constraints.NotNull",
            "title": "may not be null",
            "source": {
                "pointer": "data/attributes/name"
            },
            "meta": {
                "resourceId": "1",
                "type": "ConstraintViolation",
                "messageTemplate": "{javax.validation.constraints.NotNull.message}",
                "resourceType": "projects"
            }
        }
    ]
}
```

Notice the `422` status code used for such errors.

As mentioned above, resource validation mechanism enabled by default will be applied in case of one of the following request types: `POST`, `PUT` and `PATCH`. Once described behavior is unwanted, module should be defined in the following way:

```
{
    @Bean
    ValidationModule validationModule()
        return ValidationModule.create(false);
    }
}
```

9.3. Tracing with Zipkin/Brave

A `BraveClientModule` and `BraveServletModule` provided by `io.crnk:crnk-monitor-brave4` provides integration into Zipkin/Brave to implement tracing for your repositories. The module is applicable to both a Crnk client or server.

The Crnk 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-instrumentation-okhttp3`
- `io.zipkin.brave:brave-instrumentation-httpclient`

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

On the server-side, `BraveServletModule` 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 `BraveServletModule` does not setup tracing for incoming requests. That is the purpose of the JAX-RS/servlet integration of Brave.

Have a look at the Spring boot example application to see the `BraveServletModule` in use together with a log reporter writing the output to console.

NOTE	`io.crnk:crnk-brave` is deprecated and makes use of the Brave 3.x API.

9.4. Meta Module

This is a module that exposes the internal workings of Crnk as JSON API repositories. It lets you browse the set of available resources, their types, their attributes, etc. For example, Crnk UI make use of the meta module to implement auto-completing of input fields.

NOTE	There is currently no JSON API standard for meta data. There are more general formats like Swagger and ALPS. At some point those might be supported as well (probably rather the later than the former). One can view them to be complementary to the `MetaModule` as the later is exactly tailored towards JSON API, such as the accessability as regular JSON API (meta) repository and data structures matching the standard. Most likely, any future standard implementation will built up on the information from the `MetaModule`.

9.4.1. Setup

A setup can look as follows:

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

`ResourceMetaProvider` exposes all JSON API resources and repositories as meta data. You may add further provides to expose more meta data, such as the `JpaMetaProvider`.

9.4.2. Examples

To learn more about the set of available resources, have a look at the `MetaElement` class and all its subclasses. Some of the most important classes are:

Path	Implementation	Description
`/meta/element`	`MetaElement`	Base class implemented by any meta element.
`/meta/type`	`MetaType`	Base class implemented by any meta type element.

/meta/primitiveType	MetaPrimitiveType	Represents primitive types like Strings and Integers.
/meta/arrayType	MetaArrayType	Represents an array type.
/meta/listType	MetaListType	Represents an list type.
/meta/setType	MetaSetType	Represents an set type.
/meta/mapType	MetaMapType	Represents an map type.
/meta/dataObject	MetaDataObject	Base type for any object holding data, like JPA entities or JSON API resources.
/meta/attribute	MetaAttribute	Represents an attribute of a `MetaDataObject`.
/meta/resource	MetaResource	JSON API resource representation extending `MetaDataObject`.
/meta/resourceRepository	MetaResourceRepository	JSON API repository representation holding resources.

A `MetaResource` looks like:

```
{
    "id" : "resources.project",
    "type" : "meta/resource",
    "attributes" : {
      "name" : "Project",
      "resourceType" : "projects"
    },
    "relationships" : {
      "parent" : {
        ...
      },
      "interfaces" : {
        ...
      },
      "declaredKeys" : {
        ...
      },
      "children" : {
        ...
      },
      "declaredAttributes" : {
        ...
      },
      "subTypes" : {
        ...
      },
      "attributes" : {
        ...
      },
      "superType" : {
        ...
      },
      "elementType" : {
        ...
      },
      "primaryKey" : {
        ...
      }
    }
  }
```

A `MetaAttribute` looks like:

```
{
    "id" : "resources.project.name",
    "type" : "meta/resourceField",
    "attributes" : {
      "filterable" : true,
      "nullable" : true,
      "lazy" : false,
      "association" : false,
      "primaryKeyAttribute" : false,
      "sortable" : true,
      "version" : false,
      "insertable" : true,
      "meta" : false,
      "name" : "name",
      "updatable" : true,
      "links" : false,
      "derived" : false,
      "lob" : false,
      "cascaded" : false
    },
    "relationships" : {
      "parent" : {
        ...
      },
      "children" : {
        ...
      },
      "oppositeAttribute" : {
        ...
      },
      "type" : {
        ...
      }
    }
  }
```

9.4.3. Identifiers for Meta Elements

Of importance is the assignment of IDs to meta elements. For resources the resource type is used to compute the meta id and a `resources` prefix is added. In the example above, person gets a `resources.person` meta id. Related objects (DTOs, links/meta info) located in the same or a subpackage of a resource gets the same meta id prefix. A `ProjectData` sitting in a `dto` subpackage would get a `resources.dto.projectdata` meta id.

The meta ids are used, for example, by the Typescript generator to determine the file structure and dependencies of generated source files.

Applications are enabled to adapt the id generator process with:

```
new ResourceMetaProvider(idPrefix)
```

and

`ResourceMetaProvider.putIdMapping(String packageName, String idPrefix)`

to override the default `resources` prefix and assign a specific prefix for a package.

9.4.4. Extending the Meta Module

There is a `MetaModuleExtension` extension that allows other Crnk modules contribute `MetaProvider` implementation. This allows to:

- add `MetaFilter` implementations to intercept and modify meta elements upon initialization and request.
- add `MetaPartition` implementations to introduce new, isolated areas in the meta model, like a JPA meta model next to the JSON API one (like for documentation purposes).

For more detailed information have a look at the current `ResourceMetaProvider`.

9.5. Home Module

The `HomeModule` provides a listing of available resources in each directory (such as the root `/api/`). Note that directory paths always end with a '/' and the `HomeModule` will process the request if there is no resource or relationship repository serving that particular path.

The `HomeModule` supports two kinds of formats that can be choosen upon creation. A JSON API-style format where a links node holds all links to child directories and repositories. And a JSON HOME format as specified by JSON Home.

```
        HomeModule metaModule = HomeModule.create();
        ...
```

In the Spring Boot example applications it looks like:

```
{
  "links" : {
    "meta" : "http://localhost:8080/api/meta/",
    "projects" : "http://localhost:8080/api/projects",
    "resourcesInfo" : "http://localhost:8080/api/resourcesInfo",
    "schedule" : "http://localhost:8080/api/schedule",
    "scheduleDto" : "http://localhost:8080/api/scheduleDto",
    "tasks" : "http://localhost:8080/api/tasks"
  }
}
```

Notice the `meta` entry with a trailing '/' that allows to move to subdirectory `http://localhost:8080/api/meta/`:

```
{
  "links" : {
    "arrayType" : "http://localhost:8080/api/meta/arrayType",
    "attribute" : "http://localhost:8080/api/meta/attribute",
    "dataObject" : "http://localhost:8080/api/meta/dataObject",
    "element" : "http://localhost:8080/api/meta/element",
    "resource" : "http://localhost:8080/api/meta/resource",
    "type" : "http://localhost:8080/api/meta/type"
    ...
  }
}
```

9.6. Operations Module

By its nature RESTful applications are limited to the insertion, update and deletion of single resources. As such, developers have to design resources accordingly while having to consider aspects like transaction handling and atomicity. It is not uncommon to combine multiple data objects on the server-side and expose it as single resource to clients. It is a simple approach, but can also mean quite a substantial overhead when having to implement potentially redudant repositories. Furthermore, things like validation handling, relationships and supporting complex object graphs can get tricky when a single resource starts holding complex object graphs again.

For all the before mentioned reason support for jsonpatch.com is provided. It allows to send multiple insertions, updates and deletions with a single request and provides the results for each such executed operation. Note that numerous attempts and discussions have taken place and are still ongoing to establish a common JSON API standard, but that does not seem to make much progress. With jsonpatch.com there is already an estabilished standard that fits well for many use cases.

The implementation is provided as `OperationsModule` and the setup looks like:

```
        OperationsModule operationsModule = OperationsModule.create();
        ...
```

Further filters can be applied to intercept incoming requests. Typically applications will make use of that to start a new transaction spanning all requests. This looks as follows:

AbstractOperationsTest.java

```
            operationsModule.addFilter(new TransactionOperationFilter());
```

There is further an operations client implementation that works along the regular JSON API client implementation:

OperationsPostTest.java

```
        OperationsClient operationsClient = new OperationsClient(client);
        OperationsCall call = operationsClient.createCall();
        call.add(HttpMethod.POST, movie);
        call.add(HttpMethod.POST, person1);
        call.add(HttpMethod.POST, person2);
        call.execute();
```

An example request in JSON looks like:

```
[ {
  "op" : "POST",
  "path" : "movie",
  "value" : {
    "id" : "43e7903e-9b14-4610-96f5-69d6f2fa347d",
    "type" : "movie",
    "attributes" : {
      "title" : "test",
      ...
    },
    "relationships" : {
      ...
    }
  }
}, {
  "op" : "POST",
  "path" : "person",
  "value" : {
    "id" : "b25bfdbd-8dd6-4abd-a859-f1dedf85246b",
    "type" : "person",
    "attributes" : {
      "name" : "1",
      "version" : null
    }
  }
}, {
  "op" : "POST",
  "path" : "person",
  "value" : {
    "id" : "f9d2bda5-d2f4-4c0c-85c7-cc56b6ea91e6",
    "type" : "person",
    "attributes" : {
      "name" : "2",
      "version" : null
    }
  }
} ]
```

and an example response JSON looks as follows:

```
[ {
  "data" : {
    "id" : "43e7903e-9b14-4610-96f5-69d6f2fa347d",
    "type" : "movie",
    "attributes" : {
      "title" : "test",
      ...
    },
    "relationships" : {
      ..
    },
    "links" : {
      "self" : "http://localhost:58367/movie/43e7903e-9b14-4610-96f5-69d6f2fa347d"
    }
  },
  "status" : 201
}, {
  "data" : {
    "id" : "b25bfdbd-8dd6-4abd-a859-f1dedf85246b",
    "type" : "person",
    "attributes" : {
      "name" : "1",
      "version" : 0
    },
    "relationships" : {
      ...
    },
    "links" : {
      "self" : "http://localhost:58367/person/b25bfdbd-8dd6-4abd-a859-f1dedf85246b"
    }
  },
  "status" : 201
}, {
  "data" : {
    "id" : "f9d2bda5-d2f4-4c0c-85c7-cc56b6ea91e6",
    "type" : "person",
    "attributes" : {
      "name" : "2",
      "version" : 0
    },
    "relationships" : {
      ...
    },
    "links" : {
      "self" : "http://localhost:58367/person/f9d2bda5-d2f4-4c0c-85c7-cc56b6ea91e6"
    }
  },
  "status" : 201
} ]
```

Notice in the response a status code for each request. It is import for the `Content-Type` and `Accept` HTTP headers to have `application/json-patch+json`, otherwise the `OperationsModule` will ignore such requests.

The current limitations of the implementation are:

- So far does not support bulk `GET` operations.
- Does so far not support bulk update of relationships.

With support for `POST`, `PATCH` and `DELETE` operations the most important building blocks should be in place. The limitations are expected to be addressed at some point as well, contributions welcomed.

9.7. UI Module

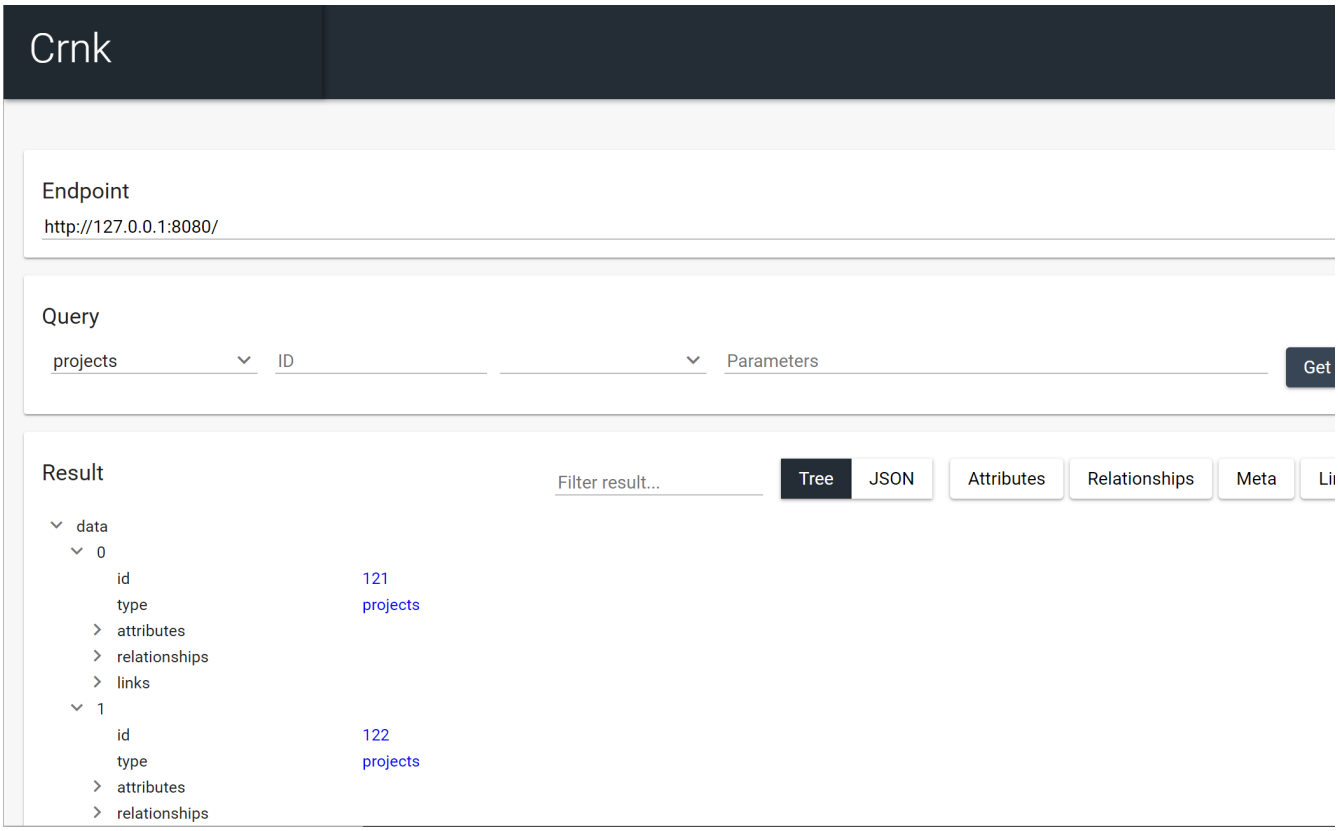
The UI module makes `crnk-ui` accessible trough the module system. It allows to browse and edit all the repositories and resources. The setup looks like:

```
        UIModule operationsModule = UIModule.create(new UIModuleConfig());
        ...
```

By default the user interface is accessible from the `/browse/` directory next to all the repositories. Have a look at the Spring Boot example application to see a working example.

This module is currently in incubation. Please provide feedback.

An example from the Spring Boot example application looks like:



9.8. Activiti Module

NOTE	This module is in new and in incubation. Feedback and improvements welcomed.

There is an `ActivitiModule` for the Activiti workflow engine that offers an alternative REST API. The motivation of `ActivitiModule` is to:

- have a JSON API compliant REST API to benefit from the resource-oriented architecture, linking, sorting, filtering, paging, and client-side tooling of JSON API.
- have a type-safe, non-generic REST API that is tailored towards the use cases at hand. This means that for each process and task definition, there is a dedicated repository and resource type for it. The resource is comprised of both the static fields provided by Activiti (like `name`, `startTime` and `priority`) and the dynamic fields stored by the application as process/task/form variables. Mapping to static resp. dynamic fields is done automatically by the `ActivitiModule` and hidden from consumers. The repository implementations ensure a proper isolation of different types. And the application is enabled, for example, to introduce custom security policies for each resource with the `SecurityModule` or a `ResourceFilter`.

This setup differs substantially from the API provided by Activiti that is implemented in generic fashion.

9.8.1. Setup

The `ActivitiModule` comes within a small example application within the `src/main/test` directory that showcases its use. It sets up an approval flow where changes to the `Schedule` resource must be approved by a user.

The `ActivitiModule` implements four resource base classes that match the equivalent Activiti classes:

- `ExecutionResource`
- `FormResource`
- `ProcessInstanceResource`
- `TaskResource`

To setup a JSON API repository for a process or task, the corresponding resource class can be subclassed and extended with the application specific fields. For example:

ApprovalProcessInstance.java

```
public abstract class ApprovalProcessInstance extends ProcessInstanceResource {

    private String resourceId;

    private String resourceType;

    public String getResourceId() {
        return resourceId;
    }

    ...
}
```

and

ScheduleApprovalProcessInstance.java

```
@JsonApiResource(type = "approval/schedule")
public class ScheduleApprovalProcessInstance extends ApprovalProcessInstance {

    private ScheduleApprovalValues newValues;

    private ScheduleApprovalValues previousValues;

    ...
}
```

The example application makes use of an intermediate `ApprovalProcessInstance` base class to potentially share the approval logic among multiple entities in the future (if it would be real-world use case). `ScheduleApprovalProcessInstance` has the static fields of Activiti and a number of custom, dynamic fields like `resourceType`, `resourceId` and `newValues`. The dynamic fields will be mapped to to process, task resp. form variables.

Notice the relation to `ApproveTask`, which is a task counter part extending from `TaskResource`. If a process has multiple tasks, you may introduce multiple such relationships.

Finally, the setup of the `ActiviModule` looks like:

ApprovalTestApplication.java

```
    public static ActivitiModule createActivitiModule(ProcessEngine processEngine) {
        ActivitiModuleConfig config = new ActivitiModuleConfig();
        ProcessInstanceConfig processConfig =
config.addProcessInstance(ScheduleApprovalProcessInstance.class);
        processConfig.filterByProcessDefinitionKey("scheduleChange");
        processConfig.addTaskRelationship(
                "approveTask", ApproveTask.class, "approveScheduleTask"
        );
        TaskRepositoryConfig taskConfig = config.addTask(ApproveTask.class);
        taskConfig.filterByTaskDefinitionKey("approveScheduleTask");
        taskConfig.setForm(ApproveForm.class);
        return ActivitiModule.create(processEngine, config);
    }
```

- `ActivitiModuleConfig` allows to register processes and tasks that then will be exposed as repositories.
- `ScheduleApprovalProcessInstance`, `ApproveTask` and the `approveTask` relationship are registered.
- `ApproveTask` is user task that is handled by submitting an `ApproveForm`.
- `filterByProcessDefinitionKey` and `filterByTaskDefinitionKey` ensure that the two repositories are isolated from other repositories for `GET`, `POST`, `PATCH` and `DELETE` operations.

One could imagine to make this configuration also available through an annotation-based API in the future as it is closely related to the resource classes and fields.

9.8.2. Example application

The example application goes a few steps further in the setup. The patterns of those steps might be of interest of consumers of the `ActivitiModule` as well.

The workflow looks as follows:

approval.bpmn20.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<definitions id="approvalDefinitions"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             targetNamespace="http://activiti.org/bpmn20"
             xmlns:activiti="http://activiti.org/bpmn"
             xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL">


    <process id="scheduleChange" name="Approve schedule change" isExecutable="true">
        <documentation>
            This process is initiated when a user modifies a scheduleEntity trough the
JSON API endpoint.
        </documentation>
```

```
        <startEvent id="startScheduleChange" name="Start"
activiti:initiator="initiator"></startEvent>

        <userTask id="approveScheduleTask" name="Approve new Schedule">
            <extensionElements>
                <activiti:formProperty id="approved" name="Do you approve this change"
type="boolean" required="true" />
            </extensionElements>
        </userTask>

        <sequenceFlow id="startFlow" sourceRef="startScheduleChange"
targetRef="approveScheduleTask"></sequenceFlow>

        <sequenceFlow id="decideFlow" sourceRef="approveScheduleTask"
targetRef="approvalExclusiveGateway"></sequenceFlow>

        <serviceTask id="scheduleChangeApproved" name="Create schedule Account, send
Alerts"

activiti:expression="${approvalManager.approved(execution)}"></serviceTask>
        <serviceTask id="scheduleChangeDenied" name="send alert"

activiti:expression="${approvalManager.denied(execution)}"></serviceTask>

        <endEvent id="endEvent" name="End"></endEvent>

        <exclusiveGateway id="approvalExclusiveGateway" name="Exclusive
Gateway"></exclusiveGateway>

        <sequenceFlow id="approveFlow" sourceRef="approvalExclusiveGateway"
targetRef="scheduleChangeApproved">
            <conditionExpression xsi:type="tFormalExpression">
                <![CDATA[
                ${approved == true}
            ]]>
            </conditionExpression>
        </sequenceFlow>

        <sequenceFlow id="denyFlow" sourceRef="approvalExclusiveGateway"
targetRef="scheduleChangeDenied">
            <conditionExpression xsi:type="tFormalExpression">
                <![CDATA[
                ${approved == false}
            ]]>
            </conditionExpression>
        </sequenceFlow>
        <sequenceFlow id="flow5" sourceRef="scheduleChangeDenied"
targetRef="endEvent"></sequenceFlow>
        <sequenceFlow id="flow6" sourceRef="scheduleChangeApproved"
targetRef="endEvent"></sequenceFlow>
```

```
    </process>
</definitions>
```

There is a:

- `approveScheduleTask` task requires a form submission by a user.
- `approvalExclusiveGateway` checks whether the change was accepted.
- `scheduleChangeApproved` invokes `${approvalManager.approved(execution)}` whereas `approvalManager` is a Java object taking care of the approval handling and registered to `activiti.cfg.xml`.
- `approvalManager.approved(⋯)` reconstructs the original request and forwards it to Crnk again to save the approved changes. This means the regular `ScheduleRepository` implementation will be called in the same fashion as for a typical request. Real world use cases may also need to save and reconstruct the security context.

For the approval-related functionality a second module is registered:

ApprovalTestApplication.java

```
    public static SimpleModule createApprovalModule(ApprovalManager approvalManager) {
        FilterSpec approvalFilter = new FilterSpec(
                Arrays.asList("definitionKey"), FilterOperator.EQ, "scheduleChange"
        );
        List<FilterSpec> approvalFilters = Arrays.asList(approvalFilter);

        SimpleModule module = new SimpleModule("approval");
        module.addRepositoryDecoratorFactory(
                ApprovalRepositoryDecorator.createFactory(approvalManager)
        );
        module.addRepository(new ApprovalRelationshipRepository(Schedule.class,
                ScheduleApprovalProcessInstance.class, "approval",
                "approval/schedule", approvalFilters)
        );
        return module;
    }
```

- `ApprovalRepositoryDecorator` hooks into the request processing of the Crnk engine and intercepts all `PATCH` and `POST` requests for the `Schedule` resource. The decorator then may chooses to abort the request and start an approval flow instead with the help of `ApprovalManager`.
- `ApprovalRelationshipRepository` introduces an additional relationship between the actual resources and approval resources. It can be used, for example, by user interfaces to show the current status of an open approval workflow. `ApprovalRelationshipRepository.getResourceFields` declares the relationship field, meaning that the original application resource does not have to declare the relationship. This may or may not be useful depending on how much control there is over the original resource (for example there is no control over JPA entities).

The chosen setup leads to an approval system that is **fully transparent** to the actual repository

implementations and can be added to any kind of repository.

`ApprovalIntTest` showcases the example workflow by doing a change, starting the approval process, submitting a form and then verifying the changes have been saved.

9.8.3. Limitations

- Currently the main entities of Activiti have been exposed. History and configuration-related repositories could be exposed as well in the future.
- Activiti has a limited query API that is inherited by the application. Potentially `crnk-jpa` could help out a bit in this area.
- Multi-tenancy is not yet done out-of-the-box.

9.9. Spring Modules and Auto Configuration

All Spring modules are hosted in the `crnk-spring` modules without explicit runtime-time dependency. Each module comes with an auto configuration that is enabled if the presence of the particular Spring component is detected. Each module comes with an enabled property to disable the auto configuration.

9.9.1. Spring MVC Module

Makes Spring MVC services available in the Crnk Home Module next to the json api repositories to have a list of all offered services. Auto configuration is provided by `CrnkSpringMvcAutoConfiguration`.

With `CrnkErrorController` configured by `CrnkErrorControllerAutoConfiguration` additionally a new error controller is provided that returns errors in JSON API format. `crnk.spring.mvc.errorController=false` allows to disable the controller.

Further integration similar to the JAX-RS one is expected in the future.

9.9.2. Spring Cloud Sleuth Module

Integrates in the same fashion as the Crnk Brave module to trace calls from a request to the individual repositories. Auto configuration is provided by `CrnkSpringCloudSleuthAutoConfiguration`.

9.9.3. Spring Security Module

`SpringSecurityModule` provides a mapping of Spring Security exception types to JSON API errors that complements the Spring-independent `SecurityModule`.

Auto configuration is provided by `CrnkSecurityAutoConfiguration`. It sets up `SecurityModule` and `SpringSecurityModule`. By default access to all repositories is blocked. A bean of type `SecurityModuleConfigurer` can be added to grant access to repositories.

9.9.4. JPA Module Auto Configuration

`CrnkJpaAutoConfiguration` setups the `JpaModule` and `SpringTransactionRunner`. `SpringTransactionRunner` let all requests run in a transaction that is automatically rolled back in case of an error. The `JpaModule` is by default setup to expose all entities as JSON API repositories. This can be disabled by setting `crnk.jpa.exposeAll=false`. The setup of the JPA module can be customized by providing a bean implementing `JpaModuleConfigurer`. See `ExampleJpaModuleConfigurer` for an example:

ExampleJpaModuleConfigurer.java

```
package io.crnk.example.springboot;


import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.criteria.CriteriaBuilder;
import javax.persistence.criteria.CriteriaQuery;
import javax.persistence.criteria.Expression;
import javax.persistence.criteria.From;

import io.crnk.example.springboot.domain.model.ScheduleDto;
import io.crnk.example.springboot.domain.model.ScheduleEntity;
import io.crnk.jpa.JpaModuleConfig;
import io.crnk.jpa.JpaRepositoryConfig;
import io.crnk.jpa.mapping.JpaMapper;
import io.crnk.jpa.query.Tuple;
import io.crnk.jpa.query.criteria.JpaCriteriaExpressionFactory;
import io.crnk.jpa.query.criteria.JpaCriteriaQueryFactory;
import io.crnk.spring.boot.JpaModuleConfigurer;
import org.springframework.stereotype.Component;

@Component
public class ExampleJpaModuleConfigurer implements JpaModuleConfigurer {

    @PersistenceContext
    private EntityManager em;

    /**
     * Expose JPA entities as repositories.
     *
     * @return module
     */

    @Override
    public void configure(JpaModuleConfig config) {
        // directly expose entity

config.addRepository(JpaRepositoryConfig.builder(ScheduleEntity.class).build());

        // additionally expose entity as a mapped dto
```

```
        config.addRepository(
                JpaRepositoryConfig.builder(ScheduleEntity.class, ScheduleDto.class,
new ScheduleMapper()).build());

        JpaCriteriaQueryFactory queryFactory = (JpaCriteriaQueryFactory)
config.getQueryFactory();

        // register a computed a attribute
        // you may consider QueryDSL or generating the Criteria query objects.
        queryFactory.registerComputedAttribute(ScheduleEntity.class, "upperName",
String.class,
                new JpaCriteriaExpressionFactory<From<?, ScheduleEntity>>() {

                    @SuppressWarnings({ "rawtypes", "unchecked" })
                    @Override
                    public Expression<String> getExpression(From<?, ScheduleEntity>
entity, CriteriaQuery<?> query) {
                        CriteriaBuilder builder = em.getCriteriaBuilder();
                        return builder.upper((Expression) entity.get("name"));
                    }
                });
    }


    class ScheduleMapper implements JpaMapper<ScheduleEntity, ScheduleDto> {

        @Override
        public ScheduleDto map(Tuple tuple) {
            ScheduleDto dto = new ScheduleDto();

            // first entry in tuple is the queried entity (if not configured
otherwise)
            ScheduleEntity entity = tuple.get(0, ScheduleEntity.class);
            dto.setId(entity.getId());
            dto.setName(entity.getName());

            // computed attribute available as additional tuple entry
            dto.setUpperName(tuple.get(1, String.class));
            return dto;
        }

        @Override
        public ScheduleEntity unmap(ScheduleDto dto) {
            // get entity from database if already there
            ScheduleEntity entity = em.find(ScheduleEntity.class, dto.getId());
            if (entity == null) {
                entity = new ScheduleEntity();
                entity.setId(dto.getId());
            }
            entity.setName(dto.getName());
            return entity;
```

```
        }

    }
}
```

9.9.5. Validation Module Auto Configuration

`CrnkValidationAutoConfiguration` sets up the `ValidationModule`. This will given validation exception to JSON API error mapping. And by default all incoming resources are validated. The later can be disabled by setting `crnk.validation.validateResources=false`.

9.9.6. UI Module Auto Configuration

`CrnkUIAutoConfiguration` sets up the `UIModule`.

9.9.7. Operations Module Auto Configuration

`CrnkOperationsAutoConfiguration` sets up the `OperationsModule`.

9.9.8. Meta Module Auto Configuration

`CrnkMetaAutoConfiguration` sets up the `MetaModule` and exposes meta data for all JSON API repositories. The `MetaModule` can be customized by providing a bean of type `MetaModuleConfigurer`.

10. Module Development

Crnk 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 crnk. The `JpaModule` in `crnk-jpa` provides a good, more advanced example of using the module API.

10.1. Request Filtering

Crnk 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 Crnk 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 Crnk 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 Crnk 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.

10.2. Resource Filtering

`ResourceFilter` allows to restrict access to resources and fields. To methods `filterResource` and `filterField` can be implemented for this purpose. Both return a `FilterBehavior` which allows to distinguish between `NONE`, `IGNORE` and `FORBIDDEN`. For example, a field like a lock count can make use of `IGNORE` in order to be ignored for POST and PATCH requests (the current value on the server is left untouched). While access to an unauthorized resource or field results in a forbidden error with `FORBIDDEN`.

The `SecurityModule` makes use of `ResourceFilter` to perform access control. `SecurityResourceFilter` in 'crnk-security` gives an example how it is used. The `MetaModule` and `HomeModule` make use of `ResourceFilterDirectory` obtained with `ModuleContext.getResourceFilterDirectory(…)` to query those `ResourceFilter` and only display information about resources and fields accessible in the context of the current re quest. The `ResourceFilterDirectory` makes use of per-request caching as the information may be accessed repeatedly for a request.

10.3. Filter Modifications

Changes to attributes and relationships can be tracked by implementing `ResourceModificationFilter`. The filter is invoked upon an incoming request while setting up the resource objects; before the actual repository is called. Such filters are useful, for example, to implement auditing functionality.

10.4. Filter Priority

`DocumentFilter`, `RepositoryFilter` and `ResourceModificationFilter` can implement `Prioritizable` to introduce a priority among multiple filters.

10.5. Access to HTTP layer

`HttpRequestContext` resp. `HttpRequestContextProvider` provides access to the HTTP requests. Most notably to get and set HTTP request and response headers. In many cases, the underlying implementation like JAXRS or Servlet provides that access as well. With `HttpRequestContext` there is an implementation that is independent of that implementation. As such it is well suited for module development, in particular for request filtering. A typical use case is to set and access security headers.

`HttpRequestContextProvider.getRequestContext` returns the request context for the currently active request. Modules have access to `HttpRequestContextProvider` trough the `ModuleContext`. Repositories, filters and modules can implement `HttpRequestContextAware` to get access to `HttpRequestContextProvider`.

10.6. Module Extensions and dependencies

ModuleExtension is an interface can can be implemented by modules to specify a contract how others can extend it. The interface has two mandator properties: targetModule and optional. targetModule specifies the module consuming those extensions (and providing the implementation for it). optional specifies whether the target module must be registered or not. In case of an optional extension without the module being registered, the extension is simply ignored. The implementing module is free to add any further, custom methods to provide extension hooks to other modules. To get access to this extensions, the module can implement ModuleExtensionAware. Extensions must be registered during Module.setupModule(...) and will be available to the target module when Module.init() is called.

For an example have a look at MetaModuleExtension and the JpaModule making use of it. The ModuleExtension was introduced with Crnk 2.0 and its use is expected to grow heavily over time.

10.7. Integrate third-party data stores

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

10.8. Implement a custom discovery mechanism

Crnk 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 Crnk integrations, like the CrnkFeature. Alternatively, it can be auto-detected through the Java service mechanism. For example, `crnk-cdi` makes use of:

META-INF/services/io.crnk.core.module.discovery.ServiceDiscovery

```
io.crnk.cdi.internal.CdiServiceDiscovery
```

Modules have access to that `ServiceDiscovery` trough the `ModuleContext.getServiceDiscovery()`.

10.9. Let a module hook into the Crnk HTTP client implementation

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

10.10. Implement a custom integration

Adding a new integration has become quite simple in recent times. Have a look at `crnk-servlet` and `crnk-rs`. Most functionality necessary is already be provided by `crnk-core`. The steps include:

- implement `HttpRequestContextBase`.
- instantiate `CrnkBoot` to setup crnk.
- get the `RequestDispatcher` from `CrnkBoot`.
- invoke the `RequestDispatcher` for each incoming request with the implemented `HttpRequestContextBase`.
- you may want to further implement `SecurityProvider`, `TransactionRunner` and `PropertiesProvider` to interface with that particular systems.

10.11. Create repositories at runtime

Repositories are usually created at compile-time, either by making use of the various annotations or a module such as the ´JpaModule´. However, the module API also allows the creation of repositories at runtime. There are two complementary mechanisms in place to achieve this and outlined in the next two sections.

NOTE	this feature is in incubation, more refinements are expected in upcoming releases.

10.11.1. Implementing repositories dynamically at runtime

There are different possibilities to implement a repository at runtime:

- Create a matching resource class at runtime with a library like http://bytebuddy.net/#/ to follow the same pattern as for any compile-time repository.
- Make use of the `Resource` class. It is the generic JSON API resource presentation within the Crnk engine.
- Make use of an arbitrary dynamic object like a `java.util.Map` and provide a `ResourceFieldAccessor` for each `ResourceField` to specify how to read and write attributes (see below for `ResourceField` examples).

In the following example we make use of the second option:

DynamicResourceRepository.java

```
public class DynamicResourceRepository extends ResourceRepositoryBase<Resource,
String> implements UntypedResourceRepository<Resource, String> {

    private static Map<String, Resource> RESOURCES = new HashMap<>();

    private final String resourceType;

    public DynamicResourceRepository(String resourceType) {
        super(Resource.class);
        this.resourceType = resourceType;
    }

    @Override
    public String getResourceType() {
        return resourceType;
    }

    @Override
    public Class<Resource> getResourceClass() {
        return Resource.class;
    }

    @Override
    public DefaultResourceList<Resource> findAll(QuerySpec querySpec) {
        return querySpec.apply(RESOURCES.values());
    }
...
}
```

This new repository can be registered to Crnk with a module:

DynamicModule

```
public class DynamicModule implements InitializingModule {

    private ModuleContext context;

    @Override
    public String getModuleName() {
        return "dynamic";
    }

    @Override
    public void setupModule(ModuleContext context) {
        this.context = context;
    }

    @Override
    public void init() {
        for (int i = 0; i < 2; i++) {
            RegistryEntryBuilder builder = context.newRegistryEntryBuilder();

            String resourceType = "dynamic" + i;
            RegistryEntryBuilder.ResourceRepository resourceRepository =
builder.resourceRepository();
            resourceRepository.instance(new DynamicResourceRepository(resourceType));

            RegistryEntryBuilder.RelationshipRepository parentRepository =
builder.relationshipRepositoryForField("parent");
            parentRepository.instance(new
DynamicRelationshipRepository(resourceType));
            RegistryEntryBuilder.RelationshipRepository childrenRepository =
builder.relationshipRepositoryForField("children");
            childrenRepository.instance(new
DynamicRelationshipRepository(resourceType));

            InformationBuilder.Resource resource = builder.resource();
            resource.resourceType(resourceType);
            resource.resourceClass(Resource.class);
            resource.addField("id", ResourceFieldType.ID, String.class);
            resource.addField("value", ResourceFieldType.ATTRIBUTE, String.class);
            resource.addField("parent", ResourceFieldType.RELATIONSHIP,
Resource.class).oppositeResourceType(resourceType)
                    .oppositeName("children");
            resource.addField("children", ResourceFieldType.RELATIONSHIP,
List.class).oppositeResourceType(resourceType)
                    .oppositeName("parent");

            context.addRegistryEntry(builder.build());
        }
    }
}
```

A new `RegistryEntry` is created and registered with Crnk. It provides information about:

- the resource and all its fields.
- the repositories and instances thereof.

Have a look at the complete example in `crnk-client` and `crnk-test` There is a further example test case and relationship repository.

10.11.2. Registering repositories at runtime

There are two possibilities to register a new repository at runtime:

- by using a `Module` and invoking `ModuleContext.addRegistryEntry` as done in the previous section.
- by implementing a `ResourceRegistryPart` and invoking `ModuleContext.addResourceRegistry`.

The first is well suited if there is a predefined set of repositories that need to be registered (like a fixed set of JPA entities in the `JpaModule`). The later is suited for fully dynamic use cases where the set of repositories can change over time (like tables in a database or tasks in an activiti instance). In this case the repositories no longer need registration. Instead the custom `ResourceRegistryPart` implementation always provides an up-to-date set of repositories that is used by the Crnk engine.

An example can be found at CustomResourceRegistryTest.java

10.12. Discovery of Modules by CrnkClient

If a module does not need configuration, it can provide a `ClientModuleFactory` implementation and register it to the `java.util.ServiceLoader` by adding a 'META-INF/services/io.crnk.client.module.ClientModuleFactory` file with the implementation class name. This lets `CrnkClient` discover the module automatically when calling `CrnkClient.findModules()`. An example looks like:

ValidationClientModuleFactory

```
package io.crnk.validation.internal;

import io.crnk.client.module.ClientModuleFactory;
import io.crnk.validation.ValidationModule;

public class ValidationClientModuleFactory implements ClientModuleFactory {

    @Override
    public ValidationModule create() {
        return ValidationModule.create();
    }
}
```

and

META-INF/services/io.crnk.client.module.ClientModuleFactory

```
io.crnk.validation.internal.ValidationClientModuleFactory
```

11. Generation

Crnk allows the generation of Typescript stubs for type-safe, client-side web development. Contributions for other languages like iOS would be very welcomed.

11.1. Typescript

The Typescript generator allows the generation of:

- interfaces for resources and related objects (like nested objects and enumeration types).
- interfaces for result documents (i.e. resources and any linking and meta information).
- interfaces for links information.
- interfaces for meta information.
- methods to create empty object instances.
- QueryDSL-like expression classes (see <expressions>)

Currently the generator targets the ngrx-json-api library. Support for other libraries/formats would be straightforward to add, contributions welcomed. A generated resource looks like:

```
import {DefaultPagedLinksInformation} from
'./information/default.paged.links.information';
import {Projects} from './projects';
import {Tasks} from './tasks';
import {CrnkStoreResource} from '@crnk/angular-ngrx';
import {
    ManyQueryResult,
    OneQueryResult,
    ResourceRelationship,
    TypedManyResourceRelationship,
    TypedOneResourceRelationship
} from 'ngrx-json-api';

export module Schedule {
    export interface Relationships {
        [key: string]: ResourceRelationship;
        task?: TypedOneResourceRelationship<Tasks>;
        lazyTask?: TypedOneResourceRelationship<Tasks>;
        tasks?: TypedManyResourceRelationship<Tasks>;
        tasksList?: TypedManyResourceRelationship<Tasks>;
        project?: TypedOneResourceRelationship<Projects>;
```

```
        projects?: TypedManyResourceRelationship<Projects>;
    }
    export interface Attributes {
        name?: string;
        description?: string;
        delayed?: boolean;
    }
}
export interface Schedule extends CrnkStoreResource {
    relationships?: Schedule.Relationships;
    attributes?: Schedule.Attributes;
}
export interface ScheduleResult extends OneQueryResult {
    data?: Schedule;
}
export module ScheduleListResult {
    export interface ScheduleListLinks extends DefaultPagedLinksInformation {
    }
    export interface ScheduleListMeta {
    }
}
export interface ScheduleListResult extends ManyQueryResult {
    data?: Array<Schedule>;
    links?: ScheduleListResult.ScheduleListLinks;
    meta?: ScheduleListResult.ScheduleListMeta;
}
export let createEmptySchedule = function(id: string): Schedule {
    return {
        id: id,
        type: 'schedule',
        attributes: {
        },
        relationships: {
            task: {data: null},
            lazyTask: {data: null},
            tasks: {data: []},
            tasksList: {data: []},
            project: {data: null},
            projects: {data: []},
        },
    };
};
```

For an example have a look at the Crnk example application, see crnk-project/crnk-example.

11.1.1. Setup

Internally the generator has to make use of the running application to gather the necessary information for generation. This approach not only supports the typical, manual implement resources and repositories manually, but also the ones obtained through third-party modules such

the JPA entities exposed by the JPA module. There are different possibilities to do that. https://github.com/crnk-project/crnk-framework/blob/master/crnk-client-angular-ngrx/build.gradle[crnk-client-angular-ngrx] does such a setup manually in Gradle. Alternatively, there is a Gradle plugin taking care of the generator setup. It makes use of the JUnit to get the application to a running state at built-time. Currently supported are CDI and Spring-based applications.

Such a setup may look like:

```
buildscript {
    dependencies {
        classpath "io.crnk:crnk-gen-typescript:${version}"
        classpath "com.moowork.gradle:gradle-node-plugin:1.1.1"
    }
}

node {
    version = '6.9.1'
    download = true
    distBaseUrl = "${ADN_NODEJS_MIRROR_BASE_URL}/dist"
}

apply plugin: 'crnk-gen-typescript'

configurations {
    typescriptGenRuntime
}

dependencies {
    typescriptGenRuntime project(':project-to-generate-from')
}

typescriptGen{

    runtime {
        configuration = 'typescriptGenRuntime'
        spring {
            profile = 'test'
            configuration =
'ch.adnovum.arch.demo.management.service.ManagementApplication'
            initializerMethod = 'someInitMethod'
            defaultProperties['someKey'] = 'someValue'
        }
    }

    npm {
        packageVersion = '0.0.1'
        packageName = '@crnk/example-api'
        description = 'example API to access example backend'
        gitRepository = 'https://.../bitbucket/scm/crnk/example.git'
        license = 'MIT'
```

```
        // packagingEnabled = false
        // outputDir = ...
    }

    includes = ['resources.task']
    excludes = ['resources.project']

    forked = true

    packageMapping['resources.something'] = '@crnk/some-other-library'
    peerDependencies['ngrx-json-api'] = '>=2.2.0'
    generateExpressions = true

    // genDir = ...

}
typescriptGen.init()
```

The basic Gradle setup makes use of:

- the `moowork` plugin is used to to gain a `node` setup.
- `crnk-meta` is used as dependency to gather a meta model of the underlying resources (or a any other type of object like JPA entities). Important to know is that every object is assigned a string-based meta id. By default the meta id matches `resources.<resourceType>`. For example a Task resource with resource type `task` has a `resources.task` meta id.
- applying `crnk-gen-typescript` results in a new `assembleTypescript` task. Consumers may want to add that task to `assemble` as dependency.

The plugin is in need for a running application to extract the set of available resources. The `runtime` section provides a number of possibilities how this can be achieved. Typically the simplest way to have a running application is to run in the same manner as for testing. This can be achieved, for example, by replicating the classpath, configuration, environment, Spring profiles, etc. Currently supported are Spring and CDI to achieve this. Please open up a ticket if you desire to work in another environment. There are a number of properties to do that:

- `runtime.configuration` sets the Gradle configuration to use to construct a classpath. In the given example `typescriptGenRuntime` is used. You may also use `compile` or anything else.
- for CDI it will work out-of-the-box if Weld is found on the classpath.
- for Spring properties allow to set the Spring profile, default properties and configuration class to use. Optionally an `initializerMethod` can be specified that is invoked before the Spring application to perform further customization.

The plugin allows various further customization options:

- `packageName`, `description`, `license`, `gitRepository` are added to the generated `package.json`.
- by default the package version matches the npm version.
- `packagingEnabled` triggers the generation of a `package.json` and `tsconfig.json` file (`true` as

default). Useful to publish the stubs to an NPM repository for use by other projects. Setting it to false will result in the generation of the Typescript files only. This may be used to generate files directly into an existing frontend project. In this case only the `generateTypescript` task is available, there is no assembly taking place.

- `genDir` specifies where source files are generated to.
- `npm.outputDirectory` specifies where the compiled NPM package should be placed (`build/npm` as default).
- `generateExpressions` specifies whether QueryDSL like classes should be generated (`false` as default).
- `packageMapping` allows to specify into which libraries resources belong to. By default all resources have a `resources.` meta id prefix and go into the currently generated package. In the example above, all resources with a `something/*` resource type resp. `resources.something` meta id prefix are included from a `@crnk/some-demo-library` library and no longer get generated manually.
- `includes` and `excludes` allow to include and exclude resources from generation based on their meta id.
- by default the generation takes place in a forked process. Since the generator typically runs the application and that may not properly cleanup, it is recommended to let the generator fork a new process to avoid resource leakage in Gradle daemons and have more stable builds.

11.1.2. Error Handling

Since the Typescript generator internally launches the application to extract information about its resources, the generator is in need of a consistent application/Crnk setup. For example, every resource must have a match repository serving it. Otherwise inconsistencies can arise that will break the generation. This means if the generation fails, it is usually best to verify the the application itself is working properly.

To track errors further down, a log file is written to `build/tmp/crnk.gen.typescript.log`. It runs with `io.crnk` on level `DEBUG` to output a large number of information.

12. Angular Development with ngrx

NOTE	this feature is still in incurbation, feedback and contributions welcomed.

This chapter is dedicated to Angular development with Crnk, ngrx and https://github.com/abdulhaq-e/ngrx-json-api[ngrx-json-api]. ngrx brings the redux-style application development from React to Angular. Its motivation is to separate the presentation layer from application state for a clean, mockable, debug-friendly, performant and scalable design.

We believe that JSON API and redux can complement each other well. The resource-based nature of JSON API and its normalized response document format (trough relationships and inclusions) are well suited to be put into an ngrx-based store. `ngrx-json-api` is a project that does exactly that. The missing piece is how to integrate Angular components like forms and tables with `ngrx-json-api`. Tables need to display JSON API resources and do sorting, filtering, paging. Forms need to display

JSON API resources and trigger `POST`, `PATCH` and `DELETE` requests. Errors should be displayed to the user in a dialog, header section or next to input component causing the issue (based on JSON API source pointers).

Crnk provides two tools: `crnk-gen-typescript` and `@crnk/angular-ngrx`. `crnk-gen-typescript` generates type-safe Typescript stubs from any Crnk backend. `@crnk/angular-ngrx` takes care of the binding of Angular forms and tables (and a few other things) to `ngrx-json-api`. `crnk-gen-typescript` and `@crnk/angular-ngrx` can be used together or individually. For more information about Typescript generation have a look at the [generation] chapter.

12.1. Feature overview

`@crnk/angular-ngrx` provides a number of different components:

Import	Description
`@crnk/angular-ngrx/operations`	`CrnkOperationsModule` implements JSON PATCH as Angular module. The module hooks into `ngrx-json-api` and enhances it with bulk insert, update, delete capabilities.
`@crnk/angular-ngrx/expression`	A simple QueryDSL-like expression model for Typescript.
`@crnk/angular-ngrx/expression/forms`	Binding of the expression model to Angular form components (a JSON API specific flavor of `ngModel`).
`@crnk/angular-ngrx/binding`	Helper classes that take care of binding tables or forms to JSON API. Makes use of `@crnk/angular-ngrx/expression`.
`@crnk/angular-ngrx/meta`	Typescript API for Meta Module generated with `crnk-gen-typescript`.
`@crnk/angular-ngrx/stub`	Some minor base classes used by Typescript generator. Not of direct interest.

All of those components are fairly lightweight and can also be used independently (if not specified otherwise above).

12.2. Bulk support with JSON Patch

`CrnkOperationsModule` imported from `@crnk/angular-ngrx/operations` provides client side support for JSON PATCH. This enables clients to issue bulk requests. See Operations module for more information about how it is implemented in Crnk.

`CrnkOperationsModule` integrates into `NgrxJsonApiModule` by replacing the implementation of `ApiApplyInitAction` in `ngrx-json-api`. Instead of issuing multiple requests, it will then issue a single bulk JSON Patch request. The bulk response triggers the usual `ApiApplySuccessAction` resp. `ApiApplyFailAction`.

Have a look at `crnk.operations.effects.spec.ts` for a test case demonstrating its use.

12.3. Expressions

`@crnk/angular-ngrx/expression` provides a QueryDSL-like expression model for Typescript. It is used to address boiler-plate when working with the Angular `FormModule` resp. `ngModel` directly. For example, when an input field needs to be bound to a JSON API resource field, a number of things must happen:

- The input field should display the current store value.
- The input field must have a unique form name.
- The input field must sent changes back to the store.
- The `FormControl` backing the input field must be properly validated. JSON API errors may may contain a source pointer. If the source pointer points to a field that is bound to a `FormControl`, it must be accounted for in its valid state.
- The input field is usually accompanied by a message field displaying validation errors.
- Errors that cannot be mapped to a `FormControl` must be displayed in a editor header or error dialog.

`ngModel` is limited to holding a simple value. In contrast, the use cases here require an understanding of the entire resource. It is necessary to have full JSON API resource and the path to the field to determine the field value and errors. This is achieved with `@crnk/angular-ngrx/expression`:

- `Expression` interface represents any kind of value that can be obtained in some fashion.
- `Path<T>` implements `Expression` and refers to a property of type `<T>` in an object.
- For nested paths like `attribute.name` two `Path` objects are nested.
- `StringPath`, `NumberPath`, `BooleanPath` and `BeanPath<T>` are type-safe implementations of path to account for primitive and `Object` types.
- `BeanBinding` implements `Path` and represents the root, usually a resource. The root has an empty path.

Such expressions and paths can be constructed manually. Or, in most cases, `crnk-gen-typescript` can take care of that. In this case usage looks like:

crnk.expression.spec.ts

```
    let bean: MetaAttribute;
    let qbean: QMetaAttribute;

    beforeEach(() => {
        bean = {
            id: 'someBean.title',
            type: 'meta/attribute',
            attributes: {
                name: 'someName'
            },
            relationships: {
                type: {
                    data: {type: 'testType', id: 'testId'},
                    reference: {type: 'testType', id: 'testId', attributes: {name:
'testName'}},
                }
            }
        };
        qbean = new QMetaAttribute(new BeanBinding(bean));
    });


    it('should bind to bean', () => {
        expect(qbean.id.getValue()).toEqual('someBean.title');
        expect(qbean.attributes.name.getValue()).toEqual('someName');
        expect(qbean.attributes.name.toString()).toEqual('attributes.name');
        expect(qbean.id.getResource()).toBe(bean);
        expect(qbean.attributes.name.getResource()).toBe(bean);
        expect(qbean.relationships.type.data.id.getValue()).toBe('testId');
        expect(qbean.relationships.type.data.type.getValue()).toBe('testType');

expect(qbean.relationships.type.reference.attributes.name.getValue()).toBe('testName')
;

expect(qbean.relationships.type.reference.attributes.name.toQueryPath()).toBe('type.na
me');
    });

    it('should update bean', () => {
        qbean.attributes.name.setValue('updatedName');
        expect(bean.attributes.name).toEqual('updatedName');
    });

    it('should provide form name', () => {

expect(qbean.attributes.name.toFormName()).toEqual('//meta/attribute//someBean.title//
attributes.name');
    });
```

Note that:

- `QMetaAttribute` from the meta model is used as example resource. At some point a dedicated test model will be setup.
- it is fully type-safe
- `getValue` fetches the value of the given path.
- `setValue` sets the value of the given path.
- `toString` returns the string representation of the path separated by dots.
- `getResource` returns the object resp. resource backing the path.
- `toFormName` computes a default (unique) form name for that path. The name is composed of the resource type, resource id and path to allow editing of multiple resources on the same screen.
- `toQueryPath` constructs a path used for sorting and filtering. Such a path does not include any `attributes`, `relationships` or `data` elements necessary to navigate through JSON API structures.
- QMetaAttribute can also be constructed without a bean binding. In this case it can still be used to construct type-safe paths and call `toString`. This can be used, for example, to specify a field for a table column where only later multiple records will then be loaded and shown.

The `CrnkBindingFormModule` provides two directives `crnkExpression` and `crnkFormExpression` that represent the `ngModel` counter-parts for expressions. While the former can be used standalone, the later is used for forms and registers itself to `ngForm` with the name provided by `toFormName`. Usage can look like:

```
<input id="nameInput" [crnkExpression]="resource.attributes.name"/>
```

or

```
<input id="nameInput" required [crnkFormExpression]="resource.attributes.name"/>
```

Notice the `required` validation directive. `crnkExpression` and `crnkFormExpression` support validation and `ControlValueAccessor` exactly like `ngModel`.

The use of expressions provides an (optional) foundation for the form and table binding discussed in the next sections.

12.4. Form Binding

Working with forms and JSON API is the same for many use cases:

- components are bound to store values
- components have to update store values by dispatching appropriate actions
- components may perform basic local validation. For example with the Angular `required` directive.

- components may get server-side validation errors using the JSON API error format.
- components may perform complex client-side validation using `@ngrx/effects`. JSON API is well suited for this purpose. For example, a (validation) effect can listen to value changes in the store and trigger `ModifyStoreResourceErrorsAction` of `ngrx-json-api` when necessary. That validation effect is free to perform any kind of validation logic cleanly decoupled from the presentation component.

The `FormBinding` class provided by `CrnkExpressionFormModule` can take care of exactly this.

12.4.1. Setup

An example setup looks like:

crnk.test.editor.component.ts

```
import {Component, OnDestroy, OnInit, ViewChild} from "@angular/core";
import {FormBinding} from "../binding/crnk.binding.form";
import {QMetaAttribute} from "../meta/meta.attribute";
import {Subscription} from "rxjs/Subscription";
import {CrnkBindingService} from "../binding/crnk.binding.service";
import {BeanBinding} from "../expression/crnk.expression";

@Component({
    selector: 'test-editor',
    templateUrl: "crnk.test.editor.component.html"
})
export class TestEditorComponent implements OnInit, OnDestroy {

    @ViewChild('formRef') form;

    public binding: FormBinding;

    public resource: QMetaAttribute;

    private subscription: Subscription;

    constructor(private bindingService: CrnkBindingService) {
    }

    ngOnInit() {
        this.binding = this.bindingService.bindForm({
            form: this.form,
            queryId: 'editorQuery'
        });

        // note that one could use the "async" pipe and "as" operator, but so
        // far code completion does not seem to work in Intellij. For this reason
        // the example sticks to slightly more verbose subscriptions.
        this.subscription = this.binding.resource$.subscribe(
            person => {
                this.resource = new QMetaAttribute(new BeanBinding(person), null);
            }
        );
    }

    ngOnDestroy() {
        this.subscription.unsubscribe();
    }
}
```

A template then looks like:

crnk.test.editor.component.html

```
<form #formRef="ngForm">
    <div *ngIf="resource != null">

        <div>
            {{binding.unmappedErrors | json}}
        </div>

        <input id="nameInput" required [crnkFormExpression]="resource.attributes.name"
/>

        <div id="valid">{{binding.valid | async}}</div>
        <div id="dirty">{{binding.dirty | async}}</div>

        <crnk-control-errors [expression]="resource.attributes.name">
            <ng-template let-errorCode="errorCode">
                <span id="controlError">{{errorCode}}</span>
            </ng-template>
        </crnk-control-errors>

        <crnk-resource-errors [expression]="resource.attributes.name">
            <ng-template let-errorData="errorData">
                <span id="resourceError">{{errorData.detail}}</span>
            </ng-template>
        </crnk-resource-errors>
    </div>
</form>
```

The `FormBinding` takes a `FormBindingConfig` as parameter. The most important parameters are `queryId` and `form`. `queryId` specifies the `ngrx-json-api` query the form is bound to, typically a query retrieving a single resource. `form` is the `NgForm` instance to interface with the Angular form mechanisms. Additionally, `zoneId` can specify in which `ngrx-json-api` zone the query is located.

12.4.2. Updating Data

`FormBinding` listens to value changes of the bound form and updates accordingly updates the store through `ngrx-json-api` actions. To have a mapping between JSON API resource fields and form controls, the later must follow a naming pattern. There are two possibilities for form control names:

- `//<type>//<id>//path` to reference a field by the resource type, resource id and path within the resource, e.g. `//person//13//attributes.name`.
- just `path` to reference a field of the resource returned by the `ngrx-json-api` query, e.g. 'attributes.name. The query must return a unique result for this to work.

The `crnkFormExpression` directive from the previous section already supports the naming schema natively. Meaning that any component making use of it, does not have to specify a name, but it will be computed based on the passed expression. This allows for type-safe development and reduces

some of the typical boiler-plate.

Note that `FormBinding` does not push changes to the store as long as local validation (`required`, `minlength`, etc.) do not pass. Two fields give access to that status information:

- `FormBinding.dirty` notifies whether bound resource(s) were modified.
- `FormBinding.valid` notifies whether bound resource(s) are invalid.

12.4.3. Validation and Error handling

`FormBinding` takes care of error handling. It maps back any JSON API errors back to the form controls of the configured form. Internally it matches the source pointers of JSON API errors against the form names (see previous section) to match errors with form control. The `FormBinding` has an `unmappedErrors` property that lists any JSON API error that could not be assigned to a specific form control instance, either because the matching instance does not exists or the JSON API error is not specific to given attribute but concerns the entire resource (like a conflict).

There are two possibilities how to display errors:

- Make use of the default Angular API and display the errors of the `FormControl` instances.
- Access the JSON API errors from the store directly.

There are two components for this purpose that work together with the expression model from the previous section. Both components take an expression as parameter:

- `crnk-control-errors` retrieves the errors from the `FormControl` having been bound to the same expression. As a consequence, it displays both local and JSON API errors.
- `crnk-resource-errors` retrieves the JSON API error directory from the store. As such, it works independently of the forms but can display JSON API errors only.

In both case a template must be specified how the error is rendered. In case of multiple errors, the template is rendered multiple times. `errorCode` and `errorData` are available as variable. `errorData` contains the full JSON API error in case of a JSON API error.

12.4.4. Roadmap and Open Issues

The Angular `FormModule` gives a number of restrictions. In the future we expect to also support the use `FormBinding` without a `NgForm` instance (for some performance and simplicity benefits). Please provide feedback in this area of what is most helpful.

WARNING	Intellij IDEA seems to have some issues when it comes to using the async pipe and code completion. For this reason the current example makes use of a subscription and avoid the async pipe.

12.5. Table Binding

Similar to `FormBinding` there is a `DataTableBinding` class. It can help taking care of interfacing a table component with JSON API.

12.5.1. Setup

An example looks like:

crnk.test.table.component.ts

```
import { Component, OnInit } from '@angular/core';
import { MetaAttributeListResult } from '../meta/meta.attribute';
import { CrnkBindingService } from '../binding/crnk.binding.service';
import { DataTableBinding } from '../binding/crnk.binding.table';
import { DataTableBindingConfig } from '../binding';
import { Observable } from 'rxjs/Observable';

@Component({
    selector: 'test-table',
    templateUrl: 'crnk.test.table.component.html'
})
export class TestTableComponent implements OnInit {

    public binding: DataTableBinding;

    public result: Observable<MetaAttributeListResult>;

    public config: DataTableBindingConfig = {
        queryId: 'tableQuery',
        fromServer: false
    }

    constructor(private bindingService: CrnkBindingService) {
    }

    ngOnInit() {
        this.binding = this.bindingService.bindDataTable(this.config);
        this.result = this.binding.result$.map(
            it => it as MetaAttributeListResult
        );
    }
}
```

and

crnk.test.table.component.html

```
<div *ngIf="result | async as result">
    <p-dataTable [value]="result.data"
                 selectionMode="single"
                 [lazy]="true" [rows]="10" [paginator]="true"
                 (onLazyLoad)="binding.onLazyLoad($event)"
                 (onRowDblclick)="open($event.data)"
    >
        <p-column field="attributes.name" [header]="name" sortable="true"
                  [filter]="true" filterMatchMode="exact">
        </p-column>
    </p-dataTable>
</div>
```

`DataTableBinding` takes a `DataTableBindingConfig` as parameters to configure the binding. Most important is the `queryId` that allows to specify which `ngrx-json-api` query should be bound to the table. `zoneId` additionally can specify in which `ngrx-json-api` zone the query is located.

12.5.2. Usage

`DataTableBinding` makes use of a `DataTableImplementationAdapter` and offers a `onLazyLoad(…)` method to translate native event of the table implementation to JSON API query parameters and then updates the query in the store accordingly. The update of the store in turns triggers a refresh from the server and lets the table component get the new data through the `result$` variable of type `Observable<ManyQueryResult>`. `ManyQueryResult` holds, next to the resources, information about linking, meta data, loading state and errors.

Note that:

- `DataTableBinding` supports the PrimeNG DataTable out-of-the-box with `DataTablePrimengAdapter`. Other tables can be supported by implementing `DataTableImplementationAdapter` and passing it as `DataTableBindingConfig.implementationAdapter`.
- `DataTableBindingConfig.customQueryParams` allows to pass custom query parameters next to the one provided by the table and initial query.
- The example is fully type-safe with the generated `MetaAttributeListResult`.

12.6. Meta Model

`@crnk/angular-ngrx/meta` hosts a Typescript API for Meta Module generated by `crnk-gen-typescript`.

13. FAQ

How to do Cors with Crnk?

In most (if not all) cases Cors should be setup in the underlying integration, like with the Servlet-API or as JAX-RS filter and not within Crnk itself. This allows to make use of the native Cors mechanisms of an integration and to share Cors handling with the other parts of the application.

Is Swagger supported by Crnk?

Have a look at http://www.crnk.io/related/.