# Crnk Cheatsheet

This documents gives a quick overview of the most important Crnk features. The document is work in progress. Please provide feedback what is desired to see.

Make sure to have read the JSON API specification to understand the concepts of resource-oriented APIs and the standards established by JSON API. Also have a look at its additional recommendations.

# 1. Setup

Crnk integrates with well with many libraries and frameworks. As such every setup might be a bit different. Have a loot at the integration examples and documentation for detailed information. In general it is easiest to setup Crnk with Spring Boot due to its auto configurations. In Gradle it can look like (see spring-boot-minimal-example):

*build.gradle*

```
buildscript {
    dependencies {
        classpath "io.spring.gradle:dependency-management-plugin:1.0.6.RELEASE"
    }
}

wrapper {
    gradleVersion = '4.10'
}

apply plugin: 'io.spring.dependency-management'
dependencyManagement {
    imports {
        mavenBom 'org.springframework.boot:spring-boot-dependencies:${SPRING_VERSION}'
        mavenBom "io.crnk:crnk-bom:${CRNK_VERSION}"
    }
}

apply plugin: 'java'

mainClassName = "io.crnk.example.service.ExampleApplication"

dependencies {
    compile "io.crnk:crnk-spring"
    compile "io.crnk:crnk-home"

    compile 'org.springframework.boot:spring-boot-starter-web'

    testCompile 'org.springframework.boot:spring-boot-starter-test'
    testCompile "io.crnk:crnk-client"
    testCompile 'com.squareup.okhttp3:okhttp:3.4.1'
}
```

# 2. Resource

A basic resource:

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

    @JsonApiId
    private Long id;

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

    ...
}
```

To rename a field:

```
@JsonProperty("alternativeName")
private String name;
```

To ignore a field:

```
@JsonIgnore
private String name;
```

To restrict access to a field:

```
@JsonApiField(sortable=false, filterable=false, postable=false, patchable=false,
readable=true)
private String name;
```

To inherit from another resource (omit `resourcePath = "task"` to let `SpecialTask` have its own url):

```
@JsonApiResource(type = "task", subTypes = SpecialTask.class)
public class Task {
    ...
}

@JsonApiResource(type = "specialTask", resourcePath = "task")
public class SpecialTask extends Task{
    ...
}
```

To nest a resource in the URL like http://example.com/posts/1/comments/2 use here.

# 3. Repositories

For more information see here.

To implement a basic repository with in-memory filter using `querySpec.apply` (or directly use `InMemoryResourceRepository`):

```
public class TaskRepository extends ResourceRepositoryBase<Task, Long> {

    private Map<Long, Task> tasks = new ConcurrentHashMap<>();

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

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

    @Override
    public <S extends Task> S create(S task) {
        task.setId(tasks.size());
        tasks.put(task.getId(), task);
        return task;
    }

    @Override
    public <S extends Task> S save(S task) {
        tasks.put(task.getId(), task);
        return task;
    }

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

# 4. Client

To query a remote JSON API repository:

```
CrnkClient client = new CrnkClient("http://localhost:8080");
ResourceRepositoryV2<Task, Serializable> taskRepo = client.getRepositoryForType(Task
.class);
QuerySpec querySpec = new QuerySpec(Task.class);
querySpec.addFilter(
  new FilterSpec(PathSpec.of("task.assignee.name"), FilterOperator.NEQ, "Joe")
);
querySpec.addSort(new SortSpec(PathSpec.of("task.assignee.name"), Direction.DESC));
querySpec.setLimit(10L);
ResourceList<Task> list = taskRepo.findAll(querySpec);
```

# 5. Relationships

More information available here

The subsequent example establishes a relationship to `Project` with the `project` field. It is backed by a primitive relationship identifier `projectId`. `projectId` is always written to the response (`SerializeType.ONLY_ID`). The `project` relationship is looked up in the project repository with `projectId` if an inclusion has been requested and the task repository did not initialize `project` (`LookupIncludeBehavior.AUTOMATICALLY_WHEN_NULL`):

```
@JsonApiRelationId
private Long projectId;

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

An eager project relationship where the project is always returned together with the task as inclusion:

```
@JsonApiRelation(serialize = SerializeType.EAGER)
private Project project;
```

To setup a relationship that is served by querying the opposite side (to get all tasks of a project, all tasks are searched for this project):

```
public class Project{
    @JsonApiRelation(
        lookUp = LookupIncludeBehavior.AUTOMATICALLY_WHEN_NULL, opposite = "project",
        repositoryBehavior = RelationshipRepositoryBehavior.FORWARD_OPPOSITE)
    private List<Tasks> tasks;
}
```

If neither a `RelationshipRepositoryBehavior` nor `@JsonApiRelationId` is applicable,

`RelationshipRepositoryV2` or rather `BulkRelationshipRepositoryV2` can implement arbitrary custom logic to lookup relationships.

# 6. Information

To attach additional (typically not persisted) information to a resource:

```
class Task{
    ...

    @JsonApiMetaInformation
    private TaskMeta meta;
}

public class TaskMeta implements MetaInformation {
    public String value;
}
```

To add additional links to resources next to the default `self` link:

```
class Task{
    ...

    @JsonApiLinksInformation
    private TaskLinks links;
}

public static class TaskLinks implements SelfLinksInformation, LinksInformation {
    private String value;
    private String self;
    ...
}
```

To attach links and meta information to lists:

```
class TaskList extends ResourceListBase<Task, TaskListMeta, TaskListLinks> {
}

class TaskListLinks implements LinksInformation {

    public String name = "value";

    ...
}

class TaskListMeta implements MetaInformation {

    public String name = "value";

    ...
}

class TaskRepository implements ... {
    ...

    @Override
    public TaskList findAll(QuerySpec querySpec){
        TaskList list = new TaskList();
        list.setMeta(...)
        list.setLinks(...)
        list.add(...)
        return list;
    }
}
```

# 7. Exception Mapping

For more information see here.

Mapped Exceptions:

- 400 `BAD_REQUEST`: `BadRequestException`
- 401 `UNAUTHORIZED`: `UnauthorizedException`
- 403 `FORBIDDEN`: `ForbiddenException`
- 404 `NOT_FOUND`: `ResourceNotFoundException`
- 405 `METHOD_NOT_ALLOWED`: `MethodNotAllowedException`
- 409 `CONFLICT`: `OptimisticLockException`
- 422 `UNPROCESSABLE_ENTITY`: `ValidationException` and `ConstraintViolationException`
- 500 `INTERNAL_SERVER_ERROR`: `InternalServerErrorException`
- 504 `GATEWAY_TIMEOUT`: `TimeoutException`

To implement a custom mapping:

*TestExceptionMapper.java*

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

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;

import java.util.List;

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;
    }
}
```

# 8. Filters/Interceptors

- `DocumentFilter` to intercept the response `Document`.
- `ResourceModificationFilter` to intercept changes to a resource.
- `ResourceFilter` to filter access to resources, e.g. to implement access control policies.
- `RepositoryDecoratorFactory` to intercept/decorate calls to repositories.

- `JpaRepositoryFilter` and the specialized `JpaCriteriaRepositoryFilter`, `QuerydslRepositoryFilter` to intercept requests to JPA repositories.

Implementations can be available to service discovery or be registered with a `Module` through `Module.setupModule(⋯)`.

# 9. Security

- Setup authentication by providing a `SecurityProvider`. JAX-RS, Spring and Servlet integration already come with an implementation out-of-the-box.
- Setup access control by using the `SecurityModule` from `crnk-security` or implement custom logic with a `ResourceFilter`. Spring Boot sets up the module automatically if found on the classpath and `SecurityConfigurer` allows its customization.

See here, a longer topic, no shortcuts here...

# 10. Data Access

There are a number of ready-to-use repository implementation that can greatly speed up development.

- Use crnk-jpa to access JPA entities.
- Use crnk-activity to access Activiti workflow and tasks.
- Use crnk-meta to get information about resources and repositories themselves.

For more information see here.