Kai Tödter

{json:api} for spring HATEOAS

Who am I?

- Distinguished Key Expert at Siemens
- Open-Source Lover & Gamer
- E-mail: kai.toedter@siemens.com
- Mastodon: https://mastodon.social/@kaitoedter
- GitHub: github.com/toedter
- code.siemens.com: kai.toedter

Show Hands!

Why JSON:API

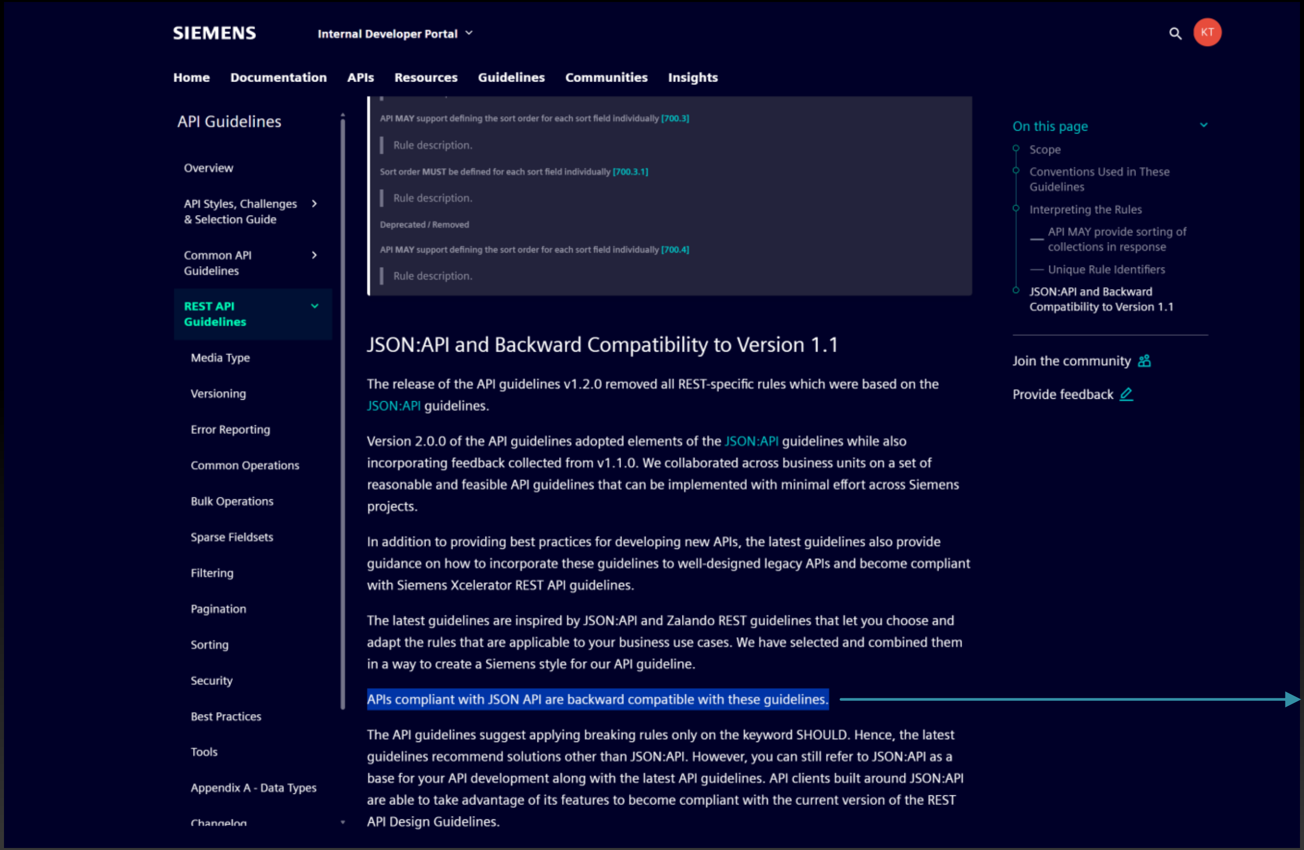
- We evaluated serveral media types / structures / frameworks for REST APIs
- JSON:API brought most the commonly needed features out of the box

JSON:API

Web site: jsonapi.org

“JSON:API is designed to minimize both the number of requests and the amount of data transmitted between clients and servers. This efficiency is achieved without compromising readability, flexibility, or discoverability.”

Siemens API Guidelines



API Guidelines

- Overview
- API Styles, Challenges & Selection Guide
- Common API Guidelines
- REST API Guidelines
  - Media Type
  - Versioning
  - Error Reporting
  - Common Operations
  - Bulk Operations
  - Sparse Fieldsets
  - Filtering
  - Pagination
  - Sorting
  - Security
  - Best Practices
  - Tools
  - Appendix A - Data Types
  - Changelog

API MAY support defining the sort order for each sort field individually [700.3]

> Rule description.

Sort order MUST be defined for each sort field individually [700.3.1]

> Rule description.

Deprecated / Removed

API MAY support defining the sort order for each sort field individually [700.4]

> Rule description.

JSON:API and Backward Compatibility to Version 1.1

The release of the API guidelines v1.2.0 removed all REST-specific rules which were based on the JSON:API guidelines.

Version 2.0.0 of the API guidelines adopted elements of the JSON:API guidelines while also incorporating feedback collected from v1.1.0. We collaborated across business units on a set of reasonable and feasible API guidelines that can be implemented with minimal effort across Siemens projects.

In addition to providing best practices for developing new APIs, the latest guidelines also provide guidance on how to incorporate these guidelines to well-designed legacy APIs and become compliant with Siemens Xcelerator REST API guidelines.

The latest guidelines are inspired by JSON:API and Zalando REST guidelines that let you choose and adapt the rules that are applicable to your business use cases. We have selected and combined them in a way to create a Siemens style for our API guideline.

APIs compliant with JSON API are backward compatible with these guidelines.

The API guidelines suggest applying breaking rules only on the keyword SHOULD. Hence, the latest guidelines recommend solutions other than JSON:API. However, you can still refer to JSON:API as a base for your API development along with the latest API guidelines. API clients built around JSON:API are able to take advantage of its features to become compliant with the current version of the REST API Design Guidelines.

On this page

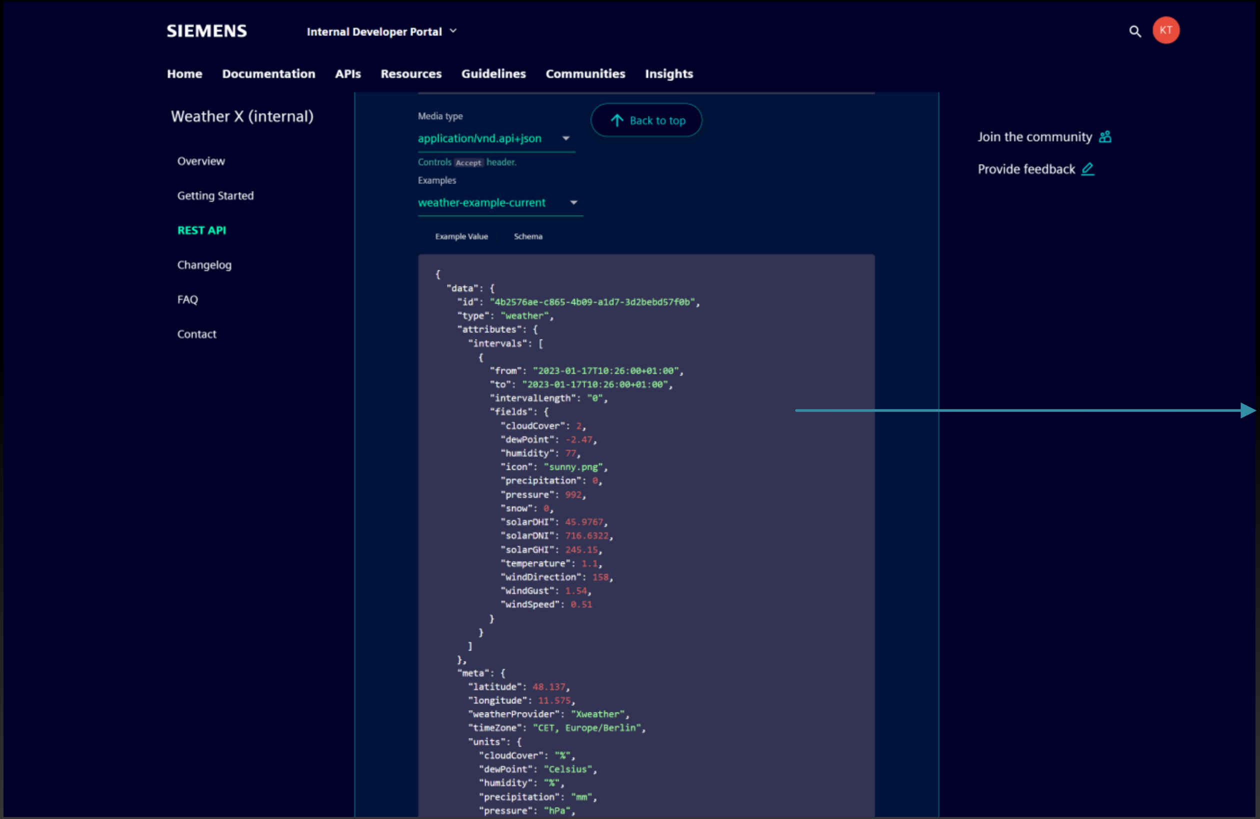
- Scope
- Conventions Used in These Guidelines
- Interpreting the Rules
  - API MAY provide sorting of collections in response
  - Unique Rule Identifiers
- JSON:API and Backward Compatibility to Version 1.1

Join the community

Provide feedback

APIs compliant with JSON:API are backward compatible with these guidelines.

Siemens Developer Portal



Media type

application/vnd.api+json

Controls `Accept` header.

Examples

weather-example-current

```
{
  "data": {
    "id": "4b2576ae-c865-4b09-a1d7-3d2bebd57f0b",
    "type": "weather",
    "attributes": {
      "intervals": [
        {
          "from": "2023-01-17T10:26:00+01:00",
          "to": "2023-01-17T10:26:00+01:00",
          "intervalLength": "0",
          "fields": {
            "cloudCover": 2,
            "dewPoint": -2.47,
            "humidity": 77,
            "icon": "sunny.png",
            "precipitation": 0,
            "pressure": 992,
            "snow": 0,
            "solarDHI": 45.9767,
            "solarDNI": 716.6322,
            "solarGHI": 245.15,
            "temperature": 1.1,
            "windDirection": 158,
            "windGust": 1.54,
            "windSpeed": 0.51
          }
        }
      ]
    },
  "meta": {
    "latitude": 48.137,
    "longitude": 11.575,
    "weatherProvider": "Xweather",
    "timeZone": "CET, Europe/Berlin",
    "units": {
      "cloudCover": "%",
      "dewPoint": "Celsius",
      "humidity": "%",
      "precipitation": "mm",
      "pressure": "hPa",
```

JSON:API example in the Siemens Developer Portal

HATEOAS

- Is for
  “Hypermedia As The Engine Of Application State”
- Very hard to pronounce ☺
- Key concept of REST
- WIKIPEDIA: With HATEOAS, a client interacts with a network application whose application servers provide information dynamically through hypermedia

Minimal JSON:API Example

```
{
  "data": {
    "id": "1",
    "type": "movies",
    "attributes": {
      "title": "The Shawshank Redemption",
      "year": 1994,
      "rating": 9.3,
      "rank": 1
    }
  },
  "links": {
    "self": "https://mymovies.com/api/movies/1"
  }
}
```

Spring HATEOAS

- Spring basic framework for REST with Hypermedia support
- Supports generic Hypermedia API
- Build-in Support for Representations like HAL, HAL-FORMS, UBER, Collection+JSON, …
- Community-based media types: JSON:API, Siren
- https://docs.spring.io/spring-hateoas/docs/current-SNAPSHOT/reference/html/

Links

Links

- Essential for hypermedia
- In REST: How to navigate to a REST resource
- Link semantic/name is called link relation
  - The relation between a REST resource and the target REST resource
- Links are well known from HTML, like
  `<a href="url">link text</a>`

Links in Spring HATEOAS

```
Link link = new Link("/my-url");
```

- A link automatically has a self relation

```
Link link = new Link("/my-url", "my-rel");
```

- A link with my-rel relation

Link Relations

- Many Link relations are standardized by IANA
  - IANA = Internet Assigned Numbers Authority
  - https://www.iana.org/assignments/link-relations/link-relations.xhtml
- Examples: self, item, next, last, …
- Recommendation: Before creating a custom name for a link relation, look up the IANA list!

Links are great!

- For providing navigation to useful other REST resources
- For providing domain knowledge to the REST clients, so that they don’t have to compute domain state on there own

Representation Models

Representation Models

- REST => Representational State Transfer
- Manipulation of resources through their representations
- Domain Model != Representation Model
- Spring HATEOAS provides RepresentationModel abstraction

Spring HATEOAS RepresentationModel

- RepresentationModel
  - Root class, for REST item resources
- CollectionModel
  - For REST collection resources
- EntityModel
  - Convenient wrapper for converting a domain model into a representation model
- PagedModel
  - Addition to CollectionModel for paged collections

Domain Model Example

```
public class Director {

    private Long id;
    private String name;

    public Director(String name) {
        this.name = name;
    }
    …
}
```

Controller Example

```
@GetMapping("/directors/{id}")
public ResponseEntity<EntityModel<Director>>
                                             findOne(@PathVariable Long id) {
   return repository.findById(id)
         .map(director -> EntityModel.of(director)
               .add(linkTo(methodOn(DirectorController.class)
                     .findOne(director.getId())).withSelfRel()))
         .map(ResponseEntity::ok)
         .orElse(ResponseEntity.notFound().build());
}
```

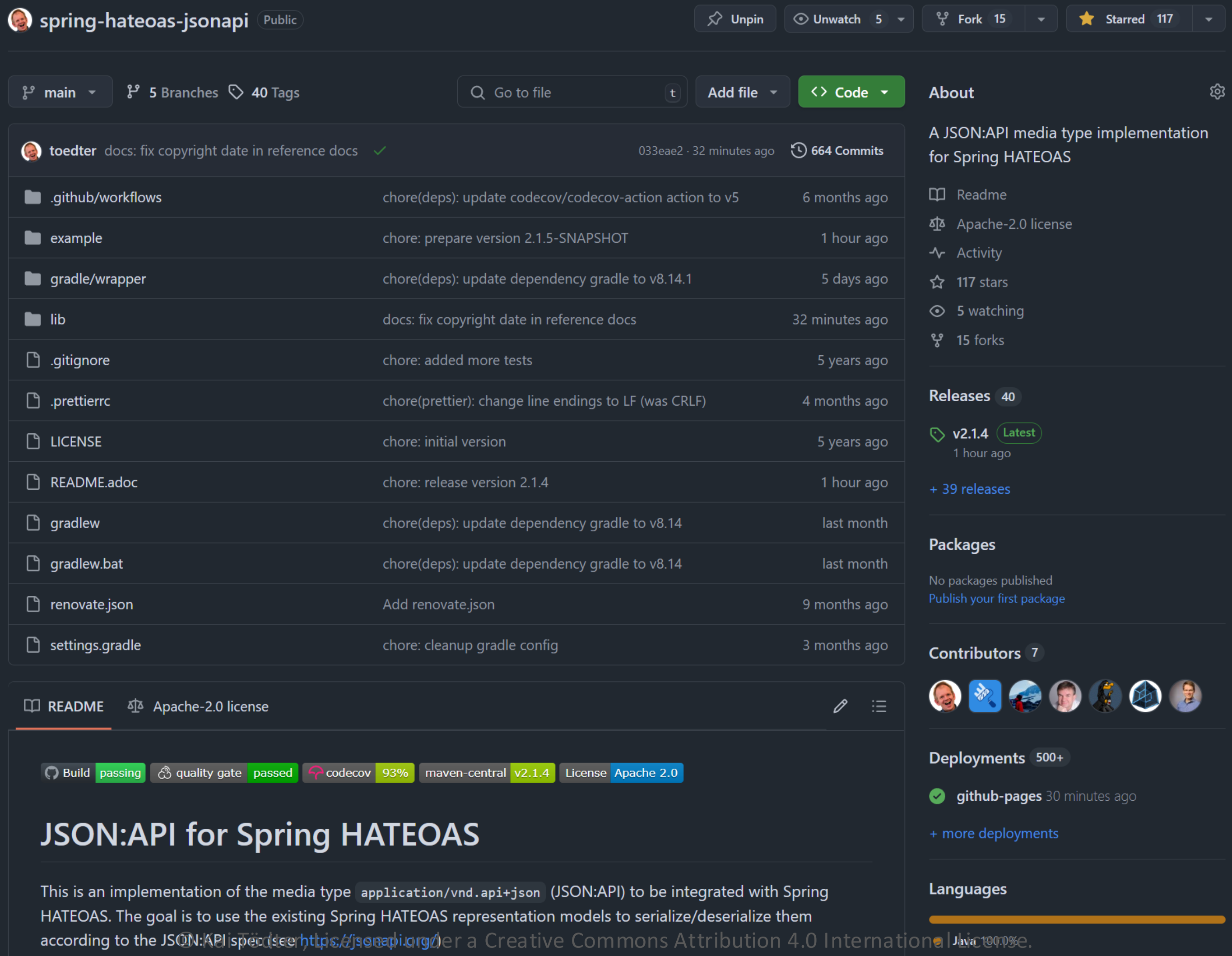
Response in HAL Media Type

```json
{
  "id": 2,
  "name": "Frank Darabont",
  "_links": {
    "self": {
      "href": "http://localhost:8080/api/directors/2"
    }
  }
}
```

{json:api} for Spring HATEOAS

JSON:API for Spring HATEOAS

- Open Source Project
- Apache 2 License
- https://github.com/toedter/spring-hateoas-jsonapi
  - Reference Documentation
  - API Documentation



spring-hateoas-jsonapi Public

toedter docs: fix copyright date in reference docs 033eae2 · 32 minutes ago 664 Commits

.github/workflows	chore(deps): update codecov/codecov-action action to v5	6 months ago
example	chore: prepare version 2.1.5-SNAPSHOT	1 hour ago
gradle/wrapper	chore(deps): update dependency gradle to v8.14.1	5 days ago
lib	docs: fix copyright date in reference docs	32 minutes ago
.gitignore	chore: added more tests	5 years ago
.prettierrc	chore(prettier): change line endings to LF (was CRLF)	4 months ago
LICENSE	chore: initial version	5 years ago
README.adoc	chore: release version 2.1.4	1 hour ago
gradlew	chore(deps): update dependency gradle to v8.14	last month
gradlew.bat	chore(deps): update dependency gradle to v8.14	last month
renovate.json	Add renovate.json	9 months ago
settings.gradle	chore: cleanup gradle config	3 months ago

JSON:API for Spring HATEOAS

This is an implementation of the media type `application/vnd.api+json` (JSON:API) to be integrated with Spring HATEOAS. The goal is to use the existing Spring HATEOAS representation models to serialize/deserialize them according to the JSON:API spec (see https://jsonapi.org/)

About

A JSON:API media type implementation for Spring HATEOAS

Releases 40

v2.1.4 Latest
1 hour ago
+ 39 releases

Packages

No packages published
Publish your first package

Contributors 7

Deployments 500+

github-pages 30 minutes ago
+ more deployments

Languages

Java 100.0%

Project Dependencies

Maven:

```
<dependency>
    <groupId>com.toedter</groupId>
    <artifactId>spring-hateoas-jsonapi</artifactId>
    <version>2.1.4</version>
</dependency>
```

Gradle:

```
implementation 'com.toedter:spring-hateoas-jsonapi:2.1.4'
```

Domain Model Example

```java
public class Director {

    private Long id;
    private String name;

    public Director(String name) {
        this.name = name;
    }
    …
}
```

Response in JSON:API media type

```
{
  "data": {
    "id": "2",
    "type": "directors",
    "attributes": {
      "name": "Frank Darabont"
    }
  },
  "links": {
    "self": "http://localhost:8080/api/directors/2"
  }
}
```

Annotations

Annotations

- @JsonApiId to mark a JSON:API id
- @JsonApiType to mark a JSON:API type
- @JsonApiTypeForClass to mark a class to provide a JSON:API type
- @JsonApiRelationships to mark a JSON:API relationship, only used for deserialization
- @JsonApiMeta to serialize/deserialize properties to JSON:API meta

Example with Annotations

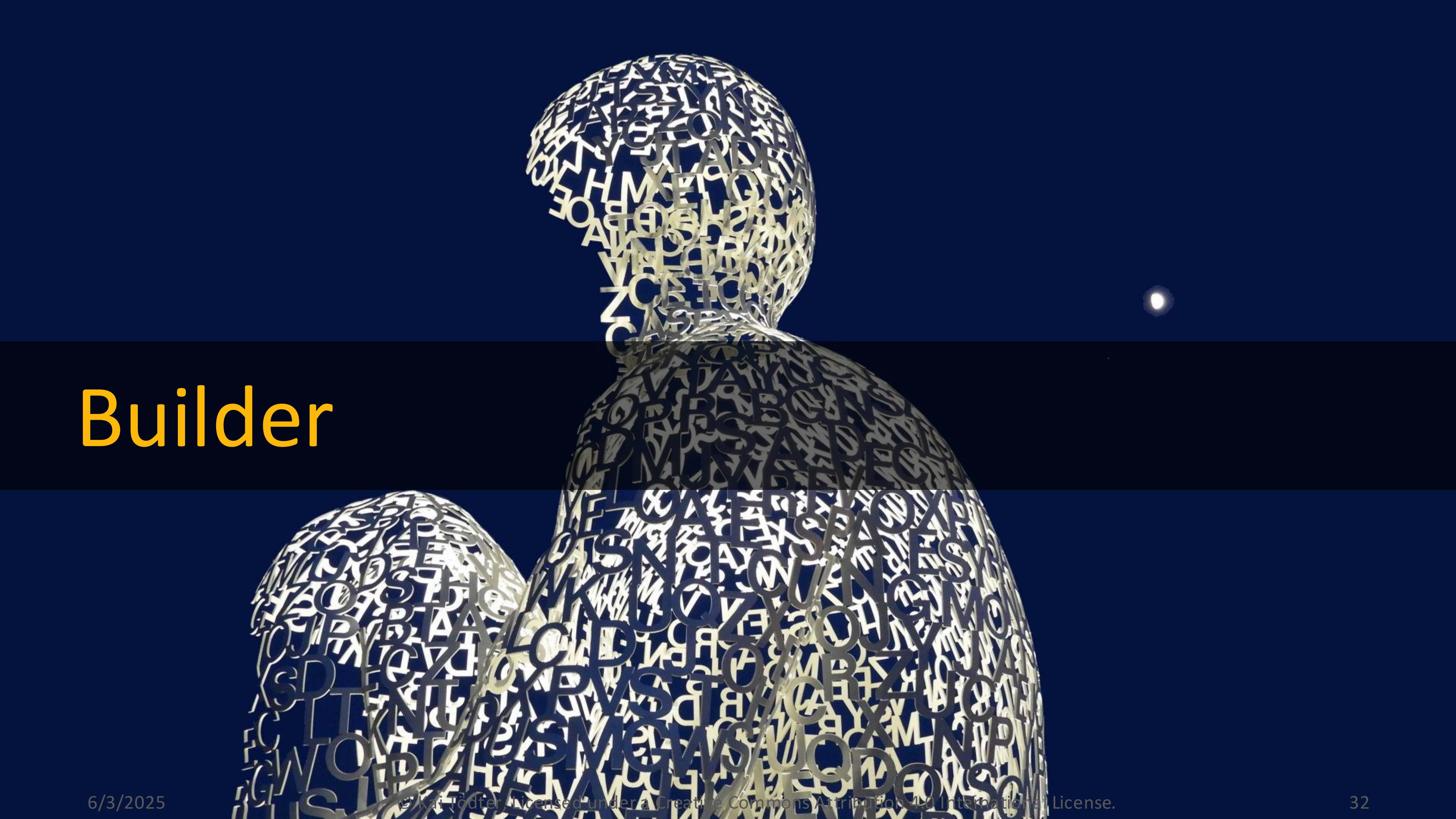
```
public class Movie {
    @Id
    private String myId;
    @JsonApiType
    private String myType;
    @JsonApiMeta
    private String myMeta;

    private String title;
}
```

Annotations Example (2)

```
EntityModel.of(
   new Movie("1", "MOVIE", "metaValue", "Star Wars"));
```

will be rendered as

```
{
   "data": {
      "id": "1",
      "type": "MOVIE",
      "attributes": {
         "title": "Star Wars"
      },
      "meta": {
         "myMeta": "metaValue"
      }
   }
}
```



Builder

JsonApiBuilder

```
Movie movie = new Movie("1", "Star Wars");

final RepresentationModel<?> jsonApiModel =
        jsonApiModel()
                .model(movie)
                .build();
```

Relationships

Relationships

- In JSON:API, relationships between REST resources are made explicit, using the relationship object
- Relationships can be to-one or to-many
- Relationships **must** contain at least one of:
    - links: a links object containing at least one of the following:
        - self: a link for the relationship itself
        - related: a related resource link
    - data: resource linkage with id and type
    - meta: meta object that contains non-standard meta-information about the relationship

Build Relationships

```
Movie movie = new Movie("1", "Star Wars");
Director director = new Director("1", "George Lucas");

final RepresentationModel<?> jsonApiModel =
    jsonApiModel()
        .model(movie)
        .relationship("directors", director)
        .build();
```

Relationsship Example

```
{
  "data": {
    "id": "1",
    "type": "movies",
    "attributes": {
      "title": "Star Wars"
    },
    "relationships": {
      "directors": {
        "data": {
          "id": "1",
          "type": "directors"
        }
      }
    }
  }
}
```

Inclusion

Inclusion of Related Resources

- With included, you can include the content of related recourses in the compound document
- The JsonApiBuilder supports adding
  - A single included resource
  - A collection of included resources
- The builder assures that included resources with same id and type appear only ONCE

Inclusion Example

```
for (Movie movie : pagedResult.getContent()) {
  jsonApiModelBuilder.included(movie.getDirectors());
}


"included": [
  {
     "id": "1",
     "type": "directors",
     "attributes": {
        "name": "Lana Wachowski"
     }
  },
...
```

Sparse Fieldsets

Convenient way to specify which

- Attributes of Resources
- Relationships (by name)
- Attributes of included Relationships

will be included in the JSON response

Controller for Sparse Fieldset

In a REST controller, a method with HTTP-mapping could provide an optional request attribute for each sparse fieldset

```
@GetMapping("/movies")
public ResponseEntity<RepresentationModel<?>> findAll(
      @RequestParam(value = "included", required = false) String[] included,
      @RequestParam(value = "fields[movies]", required = false) String[] fieldsMovies) {
```

Sparse Fieldsets Demo

Meta

- JSON:API Meta can be added using the builder or by using the @JsonApiMeta annotation
- Paging information Meta can be added automatically => Use PagedModel

Pagination Example

```
…
"links": {
   "self": "http://localhost/movies",
   "first": "http://localhost/movies?page[number]=0&page[size]=2",
   "prev": "http://localhost/movies?page[number]=0&page[size]=2",
   "next": "http://localhost/movies?page[number]=2&page[size]=2",
   "last": "http://localhost/movies?page[number]=49&page[size]=2"
},
"meta": {
   "page": {
      "number": 1,
      "size": 2,
      "totalPages": 50,
      "totalElements": 100
   }
}
```

Configuration

Configuration

You can configure

- If the JSON:API version should be rendered automatically, the default is false.
- If JSON:API types should be rendered as pluralized or non pluralized class names.
  - The default is pluralized
- If JSON:API types should be rendered as lower cased or original class names.
  - The default is lower cased
- If page information of a PagedModel should be rendered automatically as JSON:API meta object.
  - The default is true
- If a specific Java class should be rendered with a specific JSON:API type.
- A lambda expression to add additional configuration to the Jackson ObjectMapper used for serialization.
- Experimental: Render Spring HATEOAS affordances as JSON:API link meta.

Configuration Example

```
@Bean
JsonApiConfiguration jsonApiConfiguration() {
    return new JsonApiConfiguration()
            .withJsonApiVersionRendered(true)
            .withPluralizedTypeRendered(false)
            .withLowerCasedTypeRendered(false)
            .withTypeForClass(MyMovie.class, "my-movies")
            .withObjectMapperCustomizer(
                    objectMapper -> objectMapper.configure(
                            SerializationFeature.WRITE_DATES_AS_TIMESTAMPS,
                            true));
}
```

Error Handling

To create JSON:API compliant error messages, you can use JsonApiErrors and JsonApiError

```
return ResponseEntity.badRequest().body(
        JsonApiErrors.create().withError(
                JsonApiError.create()
                        .withAboutLink("http://movie-db.com/problem")
                        .withTitle("Movie-based problem")
                        .withStatus(HttpStatus.BAD_REQUEST.toString())
                        .withDetail("This is a test case")));
```

Error Example

```
{
  "errors": [
    {
      "links": {
        "about": "http://movie-db.com/problem"
      },
      "status": "400 BAD_REQUEST",
      "title": "Movie-based problem",
      "detail": "This is a test case"
    }
  ]
}
```

Conclusion

With JSON:API for Spring HATEOAS,
it is very easy to support JSON:API (serialization + deserialization) out of the box. With the builder, special JSON:API features like relationships and sparse fieldsets are supported as well.

Discussion

Links

- Spring HATEOAS: https://github.com/spring-projects/spring-hateoas
- JSON:API for Spring HATEOAS: https://github.com/toedter/spring-hateoas-jsonapi

License