Relator

A RESTful application to manage relations between persons and data based on dynamic schemas

BACHELOR THESIS

CHRISTIAN FRIES

August 2017

Thesis supervisors:

Prof. Dr. Jacques PASQUIER–ROCHA
Software Engineering Group

Pascal GREMAUD
Software Engineering Group

UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Software Engineering Group
Department of Informatics
University of Fribourg
(Switzerland)

"Productivity is never an accident. It is always the result of a commitment to excellence, intelligent planning, and focused effort."

- *Paul J. Meyer*

Abstract

Relator is a web application allowing users to manage persons and relations between persons, group persons and attach dynamic data to them in user separated workspaces. The available types of relations and dynamic data schemas can be defined by the user.

The application consists of two parts, the RESTful API called Relator API and a web client consuming the API called Relator GUI. Implementing a stable and flexible API is the main focus of this project. The GUI is a prototype acting as proof of concept.

Keywords: Web service, REST, API, JSON, PHP, Angular

Table of Contents

List of Figures

List of Tables

Listings

1 Introduction

1.1 Motivation and Goals

In the recent years, the internet has evolved tremendously. It has a huge impact on our daily lives by providing services for all different aspects of our lives. Services for medical assistance, personal fitness tracking, electronic voting, electronic banking, social networking and a lot more.

Since the introduction of REST, it has played an important role in the evolvement of the web. Every big company providing services online offers an API to interact with their content.

The main goal of this project is to understand, what the REST architecture is. To illustrate this, an app named Relator will be developed that allows users to manage persons and relations between persons, group persons and attach dynamic data to them in user separated workspaces. The available types of relations and dynamic data schemas can be defined by the user.

This main goal is split into 4 tasks:

- Study the theory of RESTful web services
- Define the domain and the architecture for the application
- Implement the RESTful API
- Implement a prototype to consume the API

1.2 Organization

Chapter 1: Introduction

The introduction contains the motivation and goals of this work, a short recapitulation of the structure of each chapter along with an overview of the formatting conventions.

Chapter 2: RESTful Web Services

This chapter explains the theoretical fundament of REST. It answers questions like what is REST, who came up with it and what are the key principles.

Chapter 3: Relator – The Architecture

This chapter introduces the domain of Relator and explains the architecture of the tool.

Chapter 4: Relator API

This chapter explains the tools that were used to implement the Relator API, how the data was modeled, what endpoints exist and some implementation specifics.

Chapter 5: Relator GUI

This chapter contains a user guide for the Relator GUI and explains how the Relator API can be accessed with another GUI.

Chapter 6: Conclusion

This chapter contains the conclusion of the project, explains the learnings of this project and what could be improved in the future.

1.3 Notations and Conventions

- The report is divided into chapters that are formatted in sections and subsections. Every section or subsection is organized into paragraphs, signaling logical breaks.
- Figures, Tables and Listings are numbered in ascending order
- Formatting conventions
  - *Italic* is used for emphasis and to signify the first use of a term.
  - **`https://api.relator.ch/v1/persons`** is used for web addresses.
  - `Monospace font` is used for class names, method names and inline code.
- Code blocks are formatted as follows

```
1     public function getName() {
2         return $this->name;
3     }
```

2 RESTful Web Services

Web services are getting more popular from day to day. In the recent years, lots of software development companies started moving from developing traditional desktop software to developing web services.

There are several reasons for that. One reason is the fact that most people today have a smartphone in their pocket and are always connected to the world wide web. Every smartphone has a web browser built in by default so, from a technical point of view, almost everyone is a potential user for a web service. Additionally, mobile devices are so powerful, they even outperform traditional computer systems in some cases.

Another reason is the simplicity to create cross-platform applications. No need for multiple code bases to support different platforms, one code base can serve all clients.

With these considerations in mind, it's important for computer scientists to understand web technologies and to keep improving them. In this report, we focus on one of them called REST.

2.1 What is REST?

The term REST, which stands for «Representational State Transfer», was introduce by Roy Fielding in the year 2000 [Fie00 76]. In his dissertation «Architectural Styles and the Design of Network-based Software-Architectures», he analyzes different styles of network-based architectures and describes a new architectural style for distributed hypermedia systems, called REST.

2.2 Key Principles

In his dissertation, Fielding describes six constraints to define the REST architecture whereof one of them is optional. These constraints are quite theoretical and apply to multiple scenarios. Bill Burke, an American software architect and author, identifies in his book «RESTful Java with JAX-RS 2.0» [Bur13] five key architectural principles for REST in the context of HTTP, so called RESTful web services.

2.2.1 Addressability

"Addressability is the idea that every object and resource in your system is reachable through a unique identifier" [Bur13 6]. For REST over HTTP, unique resource identifiers (URIs) are used as unique identifiers as they're standardized [Ber05] and widely known.

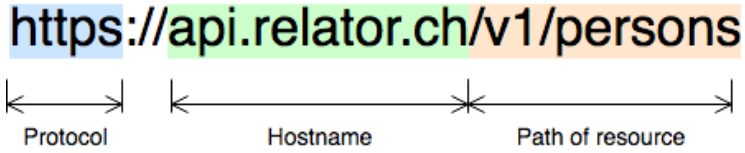


Figure 1: ***The parts of an URI***

URIs contain the protocol, the host, the port and the path to a resource in a human readable way and can be used for direct linking. If no port is specified, the default port, 80 for http and 443 for https, is used.

2.2.2 A Uniform, Constrained Interface

The idea of a uniform, constrained interface is to keep the interactions between server and client as simple as possible. To achieve this constraint, only the small set of HTTP methods is used in a RESTful web service. Each of these methods has a specific purpose as defined by the Internet Engineering Task Force (IETF) in the RFC7231 [FR14 24ff]. The most important methods are:

GET

The GET method is used for information retrieval. Browsing the internet in a web browser is mostly a sequence of GET requests sent to one or multiple servers. GET is an *idempotent* and safe operation. Idempotent means, no matter how many times you send the same request to the server, the result is always the same. Safe means in this context, that the state of the server cannot be changed by using this method.

The response to a GET request can be cached unless the *Cache-Control* header indicates otherwise.

POST

The POST method is used to create a resource on the server. The request contains a representation of the data needed to create the resource as payload. This method is non-idempotent and unsafe. Sending the same POST request multiple times will create multiple resources with different identifiers.

When a new resource is created, the response should contain a location header telling the client what URI it can use to address the newly created resource.

PUT

The PUT method is similar to the POST method but it is idempotent. While it can be used to create a resource, it's meant to be used to update a resource. The main difference to the POST method is the fact, that the PUT method needs the unique identifier of a resource. If the client can decide on the unique identifier, PUT can be used to create a resource. If not, the POST method should be used for creating and PUT should be used to replace or update a resource.

DELETE

The DELETE method is used to delete a resource. It is idempotent as well.

2.2.3 Representation-Oriented

The payload, that is transmitted between server and client, is called representation. Whatever format is used on the server to store a resource, before it is sent to the client, it is converted into a specific format of representation. Typical representations are JSON and XML but it can be any format one can come up with.

With HTTP, the representation of the resources can be negotiated between the client and the server by using a set of headers. The client specifies the representation of the request in the *Content-Type* header. By adding an *Accept* header, the client specifies its preferred response format.

2.2.4 Communicate Statelessly

In REST, stateless means that there is no client session data stored on the server. [Bur13 11] Every request is handled independently without context. If context is required, the client has to provide user state information included in the request.

This principle makes it a lot easier for RESTful web services to scale as load balancers and server clusters don't have to synchronize user state.

2.2.5 HATEOAS

HATEOAS, short for «Hypermedia as The Engine of Application State», is the final principle of REST. The idea is to not only deliver the requested resource in the response body but also, based on the application state, links to additional resources and links to interactions you can do next.

As an example, when you request a list of villages all around the world, the response could be too big and take too long to download because there are thousands of villages. The RESTful service could instead return only a list of ten villages and provide a link to the get the next set of ten villages. A request to the next set could return a link to the next set again but also to the previous set so a client can find its way back.

As another example, every village could provide a link to get all of its streets or buildings.

3 Relator – The Architecture

3.1 Use Case and Requirements

Taking pieces of information and setting them in relation creates a network of information which makes all the information within that network more valuable because they can be viewed in a bigger picture.

The idea of Relator is to manage persons and the relations between them to help you organize your environment – no matter if it's your personal environment, your business environment or a mix of both. The application shall not restrict you by providing a predefined set of relation types, it should allow you to create whatever relation you could come up with.

Identifying relations between persons allows to analyze a network of persons and how they are connected. It is not possible to attach flexible pieces of information to a person though, as a person has a predefined set of properties. Relator introduces the concept of dynamic nodes and dynamic node types to solve this restriction.

Dynamic node types allow the user to create user-defined schemas with multiple properties and multiple types of properties. Based on such a dynamic node type, dynamic nodes can be created. All the properties defined in the schema of the dynamic node type are available to store data.

This use case requires the data model to have entities providing metadata and entities to provide the actual data based on the entity providing metadata. In the Relator namespace, entities containing metadata are called node types and entities containing actual data are called nodes.

Based on this theoretical fundament, different types of nodes and different types of node types are required. A special node for persons is required, a special node for relations is required and a special node for dynamic data, a so called dynamic node, is required. To store the metadata of relations, a relation type is required and to provide a schema for dynamic nodes, a dynamic node type is required.

How these requirements can be met is described in the next chapter of this report.

3.2 Architecture

One of the big advantages of RESTful web services is separation of concerns. The user interface is separated from the data storage, allowing the two components to evolve independently [FIE00].

The Relator application is designed based on this principle. The Relator API handles the data storage and delivers the requested resources. The Relator GUI is a frontend application that consumes the Relator API.

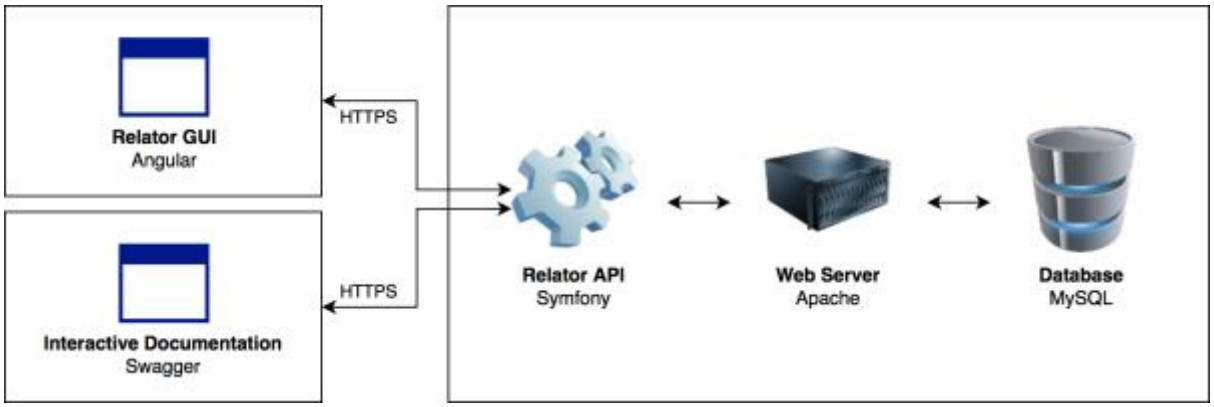


Figure 2: ***The architecture of Relator***

While the GUI does require the API to work in a meaningful way, the API is completely independent of the GUI. One could write another user interface to consume the endpoints of the API without having to know anything about the Relator GUI. As an example, the interactive documentation of this project presented in section 4.1.4 allows to browse all available endpoints of the API and test it by sending requests to the API and presenting the response.

The API is implemented in Symfony [1], a powerful and wide spread PHP framework to create web services and applications. It is served by a web server running an Apache HTTP server and a MySQL database server. Details on the implementation are described in Chapter 4.

The Relator GUI is implemented in Angular. Angular is a leading frontend development framework created and maintained by Google. It allows developing scalable applications on one codebase and reuse the code across all platforms. The first version of Angular was called «AngularJS». When the development team released the second version, they called it «Angular 2» and switched to *semantic versioning* so starting from version 4, it's officially called «Angular» only. The Relator GUI is described in Chapter 5.

4 Relator API

4.1 Implementation Tools and Services

Choosing the right set of tools to implement a piece of software is an important task. Although there are always different tools and techniques available for a specific need, choosing the right ones can make a developer's life a lot easier.

Several criterions can influence the choice such as security, technical requirements, support, price, license and the developer's knowledge.

For the implementation of Relator, I chose a set of open source software.

4.1.1 Symfony – A PHP Framework

The Relator API is based on the PHP web application framework Symfony [1] in version 3.2. Symfony was first released in 2005 under MIT license and has since been under active development. It aims to speed up the development of web application by offering a wide range of tools and presets replacing repetitive coding tasks. By implementing a lot of important design patterns such as MVC, factories and singletons and by sticking to the approach of domain-driven design, it helps writing good quality code.

Symfony encourages the use of other open source PHP projects such as PHPUnit [2]. PHPUnit is a programmer-oriented testing framework. It allows to write unit and integration test and execute them with one simple CLI command. Relator uses PHPUnit for testing the most important endpoints of the API.

Symfony's modular architecture allows developers to create extensions for the framework. Relator makes use of such extensions like the FOSUserBundle [3]. This bundle provides a basic user entity and forms to sign up, login and request a new password.

4.1.2 Doctrine – An Object-Relational Mapper

Doctrine ORM [4] is an object relational mapper for PHP. It is based on the Doctrine database abstraction layer (DBAL). The DBAL creates an interface to communicate with the database no matter what database system is used. This makes it easy for existing tools to migrate from one database system to another. On top of that, the ORM allows you to fetch objects from the database and persist objects in the database instead of writing complex SQL statements.

Doctrine provides one big benefit for PHP developers: The use of code annotations to automatically create database tables and fields. This is explained in detail in section 4.2.

4.1.3 Continuous Integration / Continuous Deployment

Continuous integration is a development practice that has gotten more popular in the recent years. The idea is to integrate code as fast as possible into a shared *repository* instead of developing code over a long period of time separated from the main repository. Repository refers to a version control system such as Git, Subversion, Maven, etc. For the development of Relator, Git is used. The main repository is hosted on Bitbucket [5].

Each commit pushed to the main repository is tested by an automated build, allowing developers to detect and solve problems early.

Continuous deployment is based on continuous integration. The idea is to deploy software to staging or production as soon as the automated tests pass successfully.

Relator uses CircleCI [6] as a service for continuous integration and continuous deployment. Every commit that is pushed to the origin on Bitbucket starts the build process on CircleCI. The build process clones the latest changes from the repository, installs required components and runs the PHPUnit tests. If they fail, CircleCI notifies registered developers that the build failed. If they succeed, the deployment process is started.

For the deployment of Relator, I use Capistrano [7], a deployment tool written in Ruby. Capistrano comes with a lot of predefined tasks for deployment so that writing a configuration for the deployment consists mostly of providing server access credentials and paths.

Depending on the branch the commit was pushed to, Capistrano uses different deployment settings. If a commit is pushed to the master branch, the deployment for the staging environment is started. If the commit is pushed to the release branch, the deployment for the production environment is started.

With this setup, creating a new release is as simple as merging changes from master branch into release branch, committing the changes and pushing them to the main repository. About 5 minutes later, the changes will be available in production.

4.1.4 Swagger – Interactive API Documentation

Swagger [8] is a set of tools to create interactive documentations for APIs. The Swagger specification defines, how the documentation has to be written in a structured way. The swagger documentation file can be created and edited in the Swagger editor by hand but it can also be generated automatically based on the code when the code is decorated with Swagger annotations. Finally, the Swagger documentation file can be visualized in the Swagger UI.

Relator is running the Swagger UI tool on `https://docs.relator.ch`.

4.2 Data model

Relator is using Doctrine ORM as noted in section 4.1.2. Doctrine creates the database schema automatically based on the entities defined in the source code. Therefore, I will mainly explain the object model and discuss only small pieces of the resulting database schema.

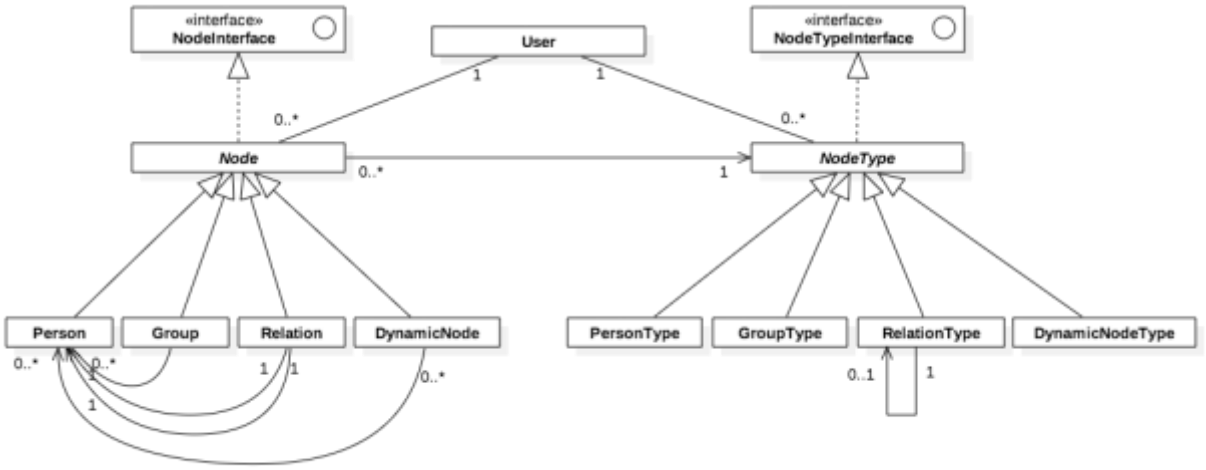


Figure 3: ***Relator domain model***

Figure 3 illustrates the domain model of the Relator API without properties and methods. The full domain model can be found in appendix B.

The data model is based on two main entities: The `Node` entity and the `NodeType` entity. All other entities of Relator extend one of these entities except for the `User` entity.

All entities of type `Node` require an associated `NodeType`. While the actual data is stored in the `Node` entity, the `NodeType` contains the metadata for the `Node`.

As the two node types `PersonType` and `GroupType` do not provide any important metadata for their nodes except for the `icon` which is the same for all the nodes of each type, these two types are not exposed through the API and are handled automatically.

The most important entities are described in the following sections.

4.2.1 User Entity

The `User` entity is one of the key entities of the Relator API. All resources of the API have a property named `owner` which is a many-to-one association to a user. Only that associated user has permission to access, modify and remove those resources.

Relator uses the FOSUserBundle [3] for the user management. This bundle provides a basic `User` entity with typical properties like `username`, `password` and `email`. The password is *hashed* and *salted* by default to maintain a high standard of security.

The `User` entity of Relator extends this basic `User` entity and adds some custom properties like the date the user was created or the date the user was modified. Like all other entities of Relator, the user entity uses the standardized *UUID format* [Lea05] as identifier.

4.2.2 NodeType Entity

The `NodeType` entity is an abstract class serving as a base class for all the node types that are described later in this chapter.

```
4     abstract class NodeType implements NodeTypeInterface
5     {
6         private $id;
7         private $title;
8         private $icon;
9         private $owner;
10        private $created;
11        private $modified;
12
13        // Setters and getters for the properties
14    }
```

Listing 1: ***Simplified version of the NodeType class***

The simplified version of the `NodeType` class presented in Listing 1 shows the properties of the class. The setters and getters are left out to save space. The abstract class implements the `NodeTypeInterface` presented in Listing 2.

```
15    interface NodeTypeInterface
16    {
17        public function getId();
18        public function getTitle();
19        public function setTitle($title);
20        public function getIcon();
21        public function setIcon($icon);
22        public function getCreated();
23        public function getModified();
24        public function getOwner();
25        public function setOwner($owner);
26    }
```

Listing 2: ***The NodeTypeInterface***

As the abstract entity `NodeType` implements the `NodeTypeInterface`, all the node types extending this class have to implement the methods defined in the interface.

The values for `id`, `created` and `modified` are generated by the persistence layer, that's why there are no setters for these properties required.

In the introduction of this section, I mentioned, that Doctrine creates the database schema directly from the source code. This can be achieved by using annotations for the class and the properties. To use a class as a Doctrine entity, the class needs the annotation `@ORM\Entity`.

```
27    /**
28     * @ORM\Entity
29     */
30    abstract class NodeType implements NodeTypeInterface {
31        // ...
32    }
```

Listing 3: ***Using a class as a Doctrine entity***

This annotation tells Doctrine to create a table for this class and use the class as an entity. To add properties as fields to the database table, they need an annotation, too. Let's have a look at the property `id`:

```
33    /**
34     * @ORM\Column(type="string", length=36)
35     * @ORM\Id
36     * @ORM\GeneratedValue(strategy="UUID")
37     */
38    private $id;
```

Listing 4: ***Using a property as a database field***

The annotation `@ORM\Column` tells Doctrine to create a database field for the property. The attributes in the parenthesis can be used to specify parameters for the database field. The annotation `@ORM\Id` tells Doctrine to mark this field as the *primary key*. The annotation `@ORM\GeneratedValue` can be used to specify the format Doctrine should use to create the value for the id.

Doctrine supports a lot of different types like strings, integers, booleans, DateTime objects etc. But what if we want to store an association to another entity? The property `owner` stores

the primary key of the user owning this `NodeType`, so let's have a closer look at the `owner` property:

```
39    /**
40     * @ORM\ManyToOne(targetEntity="User")
41     * @ORM\JoinColumn(name="owner_id", referencedColumnName="id")
42     */
43    private $owner;
```

Listing 5: ***Defining associations in Doctrine***

The property `owner` is decorated with two annotations: `@ORM\ManyToOne` and `@ORM\JoinColumn`. The first one defines the many-to-one association and uses the `User` entity as the target entity. The second one defines the name of the field to be used to store the value and what property should be used to get the value on the associated entity from.

`NodeType` is an abstract entity with the purpose of being extended by subclasses. To get this working with Doctrine, the entity needs a discriminator column. Each entity extending the `NodeType` entity needs a unique key that Doctrine can map to a specific entity.

```
44    /**
45     * @ORM\InheritanceType("JOINED")
46     * @ORM\DiscriminatorColumn(name="discr", type="string")
47     * @ORM\DiscriminatorMap({
48     *     "person"   = "PersonType",
49     *     "group"    = "GroupType",
50     *     "relation" = "RelationType",
51     *     "dynamic"  = "DynamicNodeType"
52     * })
53     */
54    abstract class NodeType implements NodeTypeInterface {
55        // ...
56    }
```

Listing 6: ***Adding a discriminator to a Doctrine entity***

The annotation `@ORM\InheritanceType` defines what type of inheritance should be used. Doctrine supports *single table inheritance* and *class table inheritance* [9]. While single table inheritance uses a single table for all the fields of all entities extending the base entity, class table inheritance uses one table for the shared fields and a separated table for each entity containing the entity-specific fields. Changing the inheritance type only affects the database, no application code has to be changed. Single table inheritance is a bit more performant as all the data is stored in one table and no joins are required. But as all fields are in one table, there are lots of empty fields. Relator is structured in a modular way and strives for a clean database structure, therefore it uses class table inheritance.

The annotation `@ORM\DiscriminatorColumn` defines the name and the type of the field to store the discriminator.

The annotation `@ORM\DiscriminatorMap` defines the keys and the according classes to use for the discriminator.

When Doctrine is told to update the database schema by calling the CLI command `./bin/console doctrine:schema:update`, the following database schema is created for the `NodeType` entity:

```
57    CREATE TABLE `nodetype` (
58      `id` varchar(36) COLLATE utf8_unicode_ci NOT NULL,
59      `owner_id` varchar(36) COLLATE utf8_unicode_ci DEFAULT NULL,
60      `title` varchar(255) COLLATE utf8_unicode_ci NOT NULL,
61      `icon` varchar(255) COLLATE utf8_unicode_ci NOT NULL,
62      `created` datetime NOT NULL,
63      `modified` datetime NOT NULL,
64      `discr` varchar(255) COLLATE utf8_unicode_ci NOT NULL,
65      PRIMARY KEY (`id`),
66      KEY `IDX_B2906CA87E3C61F9` (`owner_id`),
67      CONSTRAINT  `FK_B2906CA87E3C61F9`  FOREIGN  KEY  (`owner_id`)
      REFERENCES `user` (`id`)
68    ) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci;
```

Listing 7: ***Database schema for NodeType***

4.2.3 Node Entity

The `Node` entity is very similar to the `NodeType` entity. It is an abstract class, it implements an interface, it has a set of properties with getters and setters and uses class table inheritance for the entities extending `Node`.

To connect nodes and node types, the `Node` entity has an association to the `NodeType` entity.

```
69    /**
70     * @ORM\ManyToOne(targetEntity="NodeType")
71     * @ORM\JoinColumn(name="nodetype_id", referencedColumnName="id")
72     */
73    private $nodeType;
```

Listing 8: ***Many-to-one association from Node to NodeType***

Lifecycle methods

The `Node` class has two special properties, `title` and `icon`. These properties don't have any ORM annotations as they are not persisted to the database. They're populated by the two methods `loadTitle()` and `loadIcon()`. These two methods are so called *lifecycle callback methods*. When a `Node` object or an object extending `Node` is instantiated by the ORM, these two methods are called.

The advantage of this concept is, that every class extending the `Node` class can use its own properties and the properties of the assigned `NodeType` to define the title and the icon property.

```
74    /**
75     * @ORM\PostLoad
76     * @ORM\PostUpdate
77     */
78    public function loadIcon()
79    {
80        $this->setIcon(
81            $this->getNodeType()->getIcon()
82        );
83    }
```

Listing 9: ***Lifecycle method loadIcon() of the Node entity***

Whenever a new `Node` entity is loaded or updated, the value of the icon property is set to the value of the icon property of the assigned `NodeType`. This is handy as the icon of the node type can be changed without having to update all nodes assigned to that node type.

The generated database schema for `Node` looks like this:

```
84    CREATE TABLE `node` (
85      `id` varchar(36) COLLATE utf8_unicode_ci NOT NULL,
86      `owner_id` varchar(36) COLLATE utf8_unicode_ci DEFAULT NULL,
87      `nodetype_id` varchar(36) COLLATE utf8_unicode_ci DEFAULT NULL,
88      `created` datetime NOT NULL,
89      `modified` datetime NOT NULL,
90      `discr` varchar(255) COLLATE utf8_unicode_ci NOT NULL,
91      PRIMARY KEY (`id`),
92      KEY `IDX_857FE8457E3C61F9` (`owner_id`),
93      KEY `IDX_857FE845886D7EB5` (`nodetype_id`),
94      CONSTRAINT `FK_857FE845886D7EB5` FOREIGN KEY (`nodetype_id`)
      REFERENCES `nodetype` (`id`),
95      CONSTRAINT `FK_857FE8457E3C61F9` FOREIGN KEY (`owner_id`)
      REFERENCES `user` (`id`)
96    ) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci;
```

Listing 10: ***Database schema for Node***

4.2.4 DynamicNodeType Entity

The entity `DynamicNodeType` is the one that is used as node type for dynamic nodes. Its main purpose is to store the field configuration for the dynamic nodes assigned to that type. The configuration is provided by the user of the API and is flexible in its length and content, therefore it is not normalized in the database but stored as an array. Whenever the value is written to the database, Doctrine *serializes* the array to write it to the database. Upon retrieval, Doctrine converts the value back to an array by *un-serializing* it.

```
97    /**
98    * @ORM\Entity
99    */
100   class DynamicNodeType extends NodeType
101   {
102       /**
103        * @ORM\Column(type="array")
104        * @Assert\Type(
105        *     type="array",
106        *     message="{{ value }} is not a valid {{ type }}."
107        * )
108        */
109       private $configuration;
110
111       // Getter and setter
112   }
```

Listing 11: ***Simplified version of the DynamicNodeType entity***

The `DynamicNodeType`, like all other node types, extends the abstract entity `NodeType`. It inherits all properties and methods of its parent and has the possibility to override them.

The annotation `@ORM\Column(type="array")` tells Doctrine to automatically serialize and un-serialize the property upon saving and retrieval.

Symfony provides a set of `@Assert` annotations. These annotations can be used for validation purpose. In Listing 11, we see that the property `configuration` is decorated by the `@Assert\Type` annotation. When this class is used as a base class for a form and the value for configuration is not of type array, the form validation engine will print the validation error message defined in the annotation.

4.2.5 DynamicNode Entity

The entity `DynamicNode` is presented as an example for all the nodes of the Relator API. The purpose of the `DynamicNode` entity is to store data based on the configuration of the `DynamicNodeType` that needs to be assigned and to store the associations to the assigned `Person` entities.

```
113   /**
114    * @ORM\Entity
115    */
116   class DynamicNode extends Node
117   {
118       /**
119        * @ORM\Column(type="array")
120        */
121       private $data;
122
123       /**
124        * @ORM\ManyToMany(targetEntity="Person",
                            inversedBy="dynamicNodes")
125        * @ORM\JoinTable(
126        *  name="dynamicnode_person",
127        *  joinColumns={
128        *      @ORM\JoinColumn(name="dynamicnode_id",
                                 referencedColumnName="id")
129        *  },
130        *  inverseJoinColumns={
131        *      @ORM\JoinColumn(name="person_id",
                                 referencedColumnName="id")
132        *  }
133        * )
134        */
135       private $attachedPersons;
136
137       // Getters and setters
138   }
```

Listing 12: ***Simplified version of the DynamicNode entity***

The `data` property of the dynamic node is kept as an array and persisted as a serialized array. The property `attachedPersons` is a many-to-many association to the `Person` entity. This can be achieved by using the `@ORM\ManyToMany` annotation with the attribute `targetEntity="Person"`. The attribute `inversedBy="dynamicNodes"` defines the association as *bidirectional* and tells Doctrine what property to use on the `Person` entity to access the assigned `DynamicNode` entities. As many-to-many associations need a join table, Doctrine expects some configuration provided in the `@ORM\JoinTable` annotation.

4.3 Endpoints

The Relator API provides the endpoints listed in Table 1.

Path	Supported methods	Requires authentication
/v1/dynamic-node-types	GET, POST	Yes
/v1/dynamic-node-types/{id}	GET, PUT, DELETE	Yes
/v1/dynamic-node-types/{id}/nodes	GET	Yes
/v1/dynamic-nodes	GET, POST	Yes
/v1/dynamic-nodes/{id}	GET, PUT, DELETE	Yes
/v1/groups	GET, POST	Yes

/v1/groups/{id}	GET, PUT, DELETE	Yes
/v1/node-types	GET	Yes
/v1/nodes	GET	Yes
/v1/persons	GET, POST	Yes
/v1/persons/{id}	GET, PUT, DELETE	Yes
/v1/persons/{id}/dynamic-nodes	GET	Yes
/v1/persons/{id}/relations	GET	Yes
/v1/relation-types	GET, POST	Yes
/v1/relation-types/{id}	GET, PUT, DELETE	Yes
/v1/relations	GET, POST	Yes
/v1/relations/{id}	GET, PUT, DELETE	Yes
/v1/user/register	POST	No
/v1/user/token	POST	No

Table 1: ***Endpoints of the Relator API***

All the endpoints are documented in the interactive Swagger documentation that can be found on `https://docs.relator.ch` or in the project files presented in appendix A. In this report, I describe four important endpoints.

4.3.1 Register a New User

All endpoints of the Relator API require authentication except for the endpoint to register a new user and the endpoint to generate an access token.

Registering a new user can be achieved by sending a POST request to **`/v1/user/register`**. The Relator API accepts only the JSON format, so the request should have a `Content-Type` header set to `application/json`. Even if the header is not set, the API tries to interpret the payload as JSON data.

The endpoint expects the payload to match the schema presented in Listing 13.

```
139  {
140      "email": "mail@domain.tld",
141      "username": "username",
142      "plainPassword": {
143          "first": "password",
144          "second": "password"
145      },
146      "invitation": "12345678"
147  }
```

Listing 13: ***Example payload to create a new user***

The property `email` is required, has to be a valid email address and has to be unique. The property `username` is required and has to be unique. The properties `first` and `second` of the object `plainPassword` are required. There's two password properties to implement a «Repeat password» field. The values of `first` and `second` have to match. The property `invitation` is required and has to match the `code` property of an existing `Invitation`

record. This mechanism is implemented to restrict the public access to the Relator API. Only people with a valid invitation code can register.

If one of the properties is not valid, the API responds with the status code `400` which means `Bad request` and explains the error(s) in the body of the response:

```
{
    "errors": {
        "invitation": [
            "Your invitation code is not valid"
        ]
    },
    "status": 400,
    "type": "validation_error",
    "title": "There was a validation error"
}
```

Listing 14: ***Response to an invalid request***

Whenever an error occurs, the API responds with an `ApiProblem` that has four properties. The property `status` contains the HTTP status code. Possible codes are `400` for a validation error, `401` for an unauthorized request or `404` if a resource cannot be found. The property type contains a technical label for the error such as `validation_error`, `unauthorized` or `bad_request`. The property `title` contains a human readable error message. The property `errors` is an array and in case of a validation error, it contains all the validation error messages grouped by property. In a *graphical user interface* (*GUI*), these messages can be presented to the user.

If all properties are valid, the response to the request will contain a token that can be used to access the endpoints that require authentication. The same response is generated for successful requests to **`/v1/user/token`** that can be used to authenticate an existing user.

```
{
    "token": "eyJhbGciOiJSUzI1NiJ9.eyJyb2xlcyI6WyJST0xFX1VTS..."
}
```

Listing 15: ***Response to a successful registration or token request***

To send requests to endpoints that require authentication, the access token presented in Listing 15 has to be provided in every request header in the following format:

```
Authorization: Bearer eyJhbGciOiJSUzI1NiJ9.eyJyb2xlcyI6WyJST0xFX1VTS...
```

The Relator API uses JSON Web Tokens [10] for authentication.

4.3.2 Fetching a Collection of Dynamic Node Types

To fetch a collection of dynamic node types, the endpoint **`/v1/dynamic-node-types`** can be used. As this endpoint requires authentication, the `Authorization` header has to be set as explained in the previous section.

The response to such a request looks like this:

```
{
    "page": 1,
    "limit": 100,
    "pages": 1,
    "total": 3,
    "_links": {
        "self": {
            "href": "https://api.relator.ch/v1/dynamic-node-types?page=1&limit=100"
        },
        "first": {
            "href": "https://api.relator.ch/v1/dynamic-node-types?page=1&limit=100"
        },
        "last": {
            "href": "https://api.relator.ch/v1/dynamic-node-types?page=1&limit=100"
        }
    },
    "_embedded": {
        "items": [
            [...]
        ]
    }
}
```

Listing 16: ***Response to a GET request to an endpoint returning a collection of resources***

Every response to a GET request to an endpoint which returns a collection of resources is structured like to one presented in Listing 16. The property `page` contains the index of the current page. The property `limit` contains the number of resources that the request is limited to. The property `pages` contains to total number of pages available. The property `total` contains the number of resources that are available in total. All these properties are used for pagination which is explained in section 4.4.3.

The object `_links` contains properties that are relevant for the current state. Possible properties are `self` for the current representation, `first` for the first page of resources, `previous` or `next` for the previous or next page of resources (if multiple pages are available) and `last` for the last page of resources.

The property `_embedded` is an object which contains all the requested resources in a property called `items`.

In case of dynamic node types, such an item looks like this:

```
{
  "id": "fd7c5b2c-81d9-11e7-be10-d077e30d96ca",
  "title": "Meeting",
  "icon": "icon-node",
  "created": "2017-08-15T18:51:31+02:00",
  "modified": "2017-08-15T18:51:31+02:00",
  "configuration": {
    "fields": [
      {
        "name": "topic",
        "label": "Topic",
        "type": "TextField",
        "options": {
          "required": true
        }
      },
      {
        "name": "date",
        "label": "Date",
        "type": "DateField",
        "options": {
          "required": true
        }
      },
      {
        "name": "notes",
        "label": "Notes",
        "type": "TextareaField",
        "options": {
          "required": true
        }
      }
    ]
  },
  "discr": "dynamic",
  "_links": {
    "self": {
      "href": "http://api.relator.dev/v1/dynamic-node-
types/fd7c5b2c-81d9-11e7-be10-d077e30d96ca"
    }
  }
}
```

Listing 17: ***One item of the collection returned by GET /v1/dynamic-node-types***

The item presented in Listing 17 is a JSON representation of the entity `DynamicNodeType`. Representations can look different based on their contexts. This is explained in section 4.4.1.

The interesting part is the property `configuration`. It contains the configuration for the data structure of dynamic nodes using this dynamic node type. It contains a property called `fields`. Every object in the `fields` array represents one property that the dynamic node can or has to have, depending on the sub-property `options.required` being true or false. The structure of the fields object is explained in detail in the next section.

4.3.3 Creating a Dynamic Node Type

To create a new dynamic node type, a `POST` request with a valid `Authorization` header can be sent to **`/v1/dynamic-node-types`**. The payload should look as presented in Listing 18.

```
{
    "title": "Custom dynamic node type",
    "icon": "icon-node",
    "configuration": {
      "fields": [
        {
          "name": "title",
          "label": "The title of the dynamic node",
          "type": "TextField",
          "options": {
            "required": 1
          }
        },
        {
          "name": "brand",
          "label": "A brand for the dynamic node",
          "type": "SelectField",
          "options": {
            "required": 1,
            "options": ["Armani", "Burberry", "Hugo Boss"]
          }
        }
      ]
    }
}
```

Listing 18: ***Payload to create a new dynamic node type***

The property `title` is used as the title for the dynamic node type. The property `icon` contains an identifier for an icon that can be used to represent this type. The property `configuration` contains the configuration for the fields.

Every object in the `fields` array defines one property for the dynamic nodes assigned to this dynamic node type. Table 2 explains the available properties.

Property	Explanation
`name`	The name to use for the property on the dynamic node. This name should not contain special chars.
`label`	The user readable label for the field. In a GUI, this can be used as label for the form field.
`type`	The type of the field to use for this property. The following types are available: `TextField` for single-line text `TextareaField` for multi-line text `DateField` for a date `SelectField` for the choice of predefined values provided in the `options` property of the `options` object

`options`	The options property is an object and knows these sub-properties: `required`: Defines if a property is mandatory or optional `options`: This property is only used for the type `SelectField`. As options, you can provide an array of strings to make them available as choices.

Table 2: ***Available properties for the field configuration of a dynamic node type***

If all properties are valid, the API will respond with the status code `201` which means `Created`. A JSON representation of the newly created resource can be found in the response body. Additionally, the response contains a `Location` header with the URI of the newly created resource. The JSON representation looks like the one described in Listing 17, it contains the properties `id, created, modified` and `_links` added by the server.

4.3.4 Creating a Dynamic Node

To demonstrate how dynamic nodes can be created based on a dynamic node type, let's create a dynamic node based on the dynamic node type created in the previous section.

To create a dynamic node, we send a `POST` request to **`/v1/dynamic-nodes`** with a valid `Authorization` header.

The example payload presented in Listing 19 demonstrates a valid representation:

```
249  {
250      "data": {
251          "title": "The title of the dynamic node",
252          "brand": "Hugo Boss"
253      },
254      "attachedPersons": [
255          "id-of-first-person-to-attach",
256          "id-of-second-person-to-attach"
257      ],
258      "nodeType": "id-of-dynamic-node-type"
259  }
```

Listing 19: ***Payload to create a new dynamic node***

The property `data` is an object and expects data based on the dynamic node type. Each property name of the `data` object has to match with the `name` property of a `field` configured on the dynamic node type. If the `data` object contains a property that is not defined in the dynamic node type configuration, the API will respond with a HTTP status `400` and a validation error message. The same applies if the `data` object doesn't contain a property that is defined as required or a property of type `SelectField` has a value which is not defined in the `options` array of the `options` object.

4.4 Implementation Specifics

A lot of code was written to implement the Relator API. In this section, I explain three important parts.

4.4.1 Transforming Entities to Representations

In section 2.2.3 we learned about the representation oriented principle of REST. Big APIs allow the client to request different formats of representations such as JSON or XML. The Relator API supports only the JSON format.

Internally, the entities are stored in a relational MySQL database. Doctrine fetches that data and maps it to entity objects. To respond to a request, the entities have to be transformed into a JSON representation. This transformation is performed by the *serializer* of the JMSSerializerBundle [11].

The JMSSerializerBundle allows to transform Doctrine entities and other objects into JSON representations based on a simple configuration.

```
260  Relator\ApiBundle\Entity\Nodes\DynamicNode:
261    exclusion_policy: ALL
262
263    properties:
264      data:
265        expose: true
266      attachedPersons:
267        expose: true
```

Listing 20: ***Basic serializer configuration***

The configuration in Listing 20 presents a basic configuration for the `DynamicNode` entity in the *YAML* format. The property `exclusion_policy` can be used to define if by default all properties should be exposed or not. For security reasons, the Relator API excludes all properties by default. When a new property is added to an entity, it won't be exposed to the public until the configuration for the property is set to `expose: true` as set for the properties `data` and `attachedPersons`.

A big advantage of using the JMSSerializerBundle is the possibility to use *virtual properties*. The bundle allows you to define methods that are executed on serialization and the return values of the methods is added to the representation as if it were part of the resource.

```
268  Relator\ApiBundle\Entity\Nodes\DynamicNode:
269    virtual_properties:
270      getDynamicNodeConfiguration:
271        serialized_name: configuration
```

Listing 21: ***Virtual properties in the serializer configuration***

Listing 21 presents a virtual property defined in the serializer configuration of `DynamicNode`. When a `DynamicNode` entity is serialized, the method `getDynamicNodeConfiguration()` is executed and the return value of that method is added to the representation as property `configuration`. The Relator API uses this method to provide the configuration of the `DynamicNodeType` as a property on the `DynamicNode`.

Different representations based on the context

Sometimes you need to expose properties based on the current context. Imagine you have an entity with a lot of properties. Some of them might be very important so they always have to

be part of the representations. Other might be less relevant and should only be exposed when the resource is requested directly but not when the resource is embedded in a collection of resources. This can be handled by defining different serialization groups.

Let's have a look at an excerpt of the serializer configuration of `Person`:

```
272   Relator\ApiBundle\Entity\Nodes\Person:
273     properties:
274       familyName:
275         expose: true
276         groups: [list, detail]
277       nickname:
278         expose: true
279         groups: [detail]
```

Listing 22: ***Excerpt of the serializer configuration of Person***

Every property in the serializer configuration has defined an array of groups. The property `familyName` is key for a person so it is added to the groups `list` and `detail`. The property `nickname` is less important so it is added only to the group `detail`. When a collection of persons is requested by sending a `GET` request to **`/v1/persons`**, the `ApiController` will call the serializer and tell him to include only properties of the group `list`. In this case, the property `nickname` will not be included in the representation. When a single resource is requested by sending a `GET` request to **`/v1/persons/{id}`**, the `ApiController` will call the serializer and tell him to include all properties of the group `detail`. In this case, both properties `familyName` and `nickname` will be included.

4.4.2 Adding Links to Representations

In section 2.2.5 we learned about HATEOAS and the importance of linking resources based on the context. The Relator API generates context dependant links based on the BazingaHateoasBundle [12].

The links can be configured in the same way as the serializer configuration described in the previous section.

```
280   Relator\ApiBundle\Entity\Nodes\Person:
281     relations:
282       -
283         rel: relations
284         href:
285           route: relator_api_persons_list_relations
286           absolute: true
287           parameters:
288             id: expr(object.getId())
289
290       -
291         rel: dynamic-nodes
292         href:
293           route: relator_api_persons_list_dynamicnodes
294           absolute: true
295           parameters:
296             id: expr(object.getId())
```

Listing 23: ***Excerpt of HATEOAS configuration of the Person entity***

The links for an entity can be configured in the `relations` section. Every relation requires the two properties `rel` and `href`. The property `rel` defines the property name that is used in the `_links` section of the representation and the property `href` defines the URI. Relator uses the Symfony built-in routing component to generate URIs.

When a representation of a `Person` entity is requested, the links are available in the `_links` object of the response body:

```
297   {
298       [...]
299       "_links": {
300           "relations": {
301               "href": "http://api.relator.dev/v1/persons/fd7f2410-
      81d9-11e7-be10-d077e30d96ca/relations"
302           },
303           "dynamic-nodes": {
304               "href": "http://api.relator.dev/v1/persons/fd7f2410-
      81d9-11e7-be10-d077e30d96ca/dynamic-nodes"
305           }
306       }
307   }
```

Listing 24: ***Example HATEOAS links of a Person representation***

These links can be used to get further information about the person like the relations or the dynamic nodes attached to that person.

4.4.3 Collections

If a response contains more than one resource in the response body, that group of resources is called a collection. For collections, some special rules apply.

Pagination

All collections of the Relator API use pagination. The pagination works by using two parameters, `page` and `limit`. The `limit` parameter defines, how many resources to return on

one page and the `page` parameter defines the index of the page to return. By default, the `limit` is set to `100`.

https://api.relator.ch/v1/api/persons?limit=20&page=2

This URI returns 20 `Person` resources with an offset of `20`, meaning that the second 20 `Person` resources are returned.

If there is a total of 35 resources and a consumer requests `page=3`, the API will respond with an exception saying that the page does not exist.

Sorting

By default, collections are ordered by modification date in descending order. The consumer of the API can change the order property and the order direction by adding a `sorting` parameter.

https://api.relator.ch/v1/api/persons?sorting=familyName

This URI will return a collection of resources sorted by `familyName` in ascending order. To change the order direction, you can set the order direction by adding a colon after the sorting property and the desired order direction:

https://api.relator.ch/v1/api/persons?sorting=familyName:desc

This URI will return a collection of `Person` resources sorted by `familyName` in descending order.

It is also possible to sort by multiple properties by defining a comma separated list:

https://api.relator.ch/v1/api/persons?sorting=familyName,givenName:desc

By default, every entity can be sorted by the properties `icon`, `created` and `modified`. For entities extending NodeType, an additional property `title` is available.

For every entity it is defined, which of its properties are sortable by implementing a static method `getSortableProperties()`.

```
308   public static function getSortableProperties() {
309       $defaultFields = parent::getSortableProperties();
310       return array_merge([
311           'start',
312           'end'
313       ], $defaultFields);
314   }
```

Listing 25: ***Definition of sortable fields of an entity***

Listing 25 shows the implementation of `getSortableProperties()` of the entity `Relation`. This entity can be sorted by the default properties of a node which are defined on the node itself and can be accessed by calling `parent::getSortableProperties()`. Additionally, the two properties `start` and `end` are added.

Filtering

The endpoint **`/v1/persons`** allows filtering of the `Person` resources by adding a `search` parameter. This feature can be used for auto-completion.

The searchable properties are defined on the `Person` entity, the same way as the sortable properties are defined, and can be accessed by calling the static method `getSearchableProperties()`.

```
315   public static function getSearchableProperties() {
316       $defaultFields = parent::getSearchableProperties();
317       return array_merge([
318           'givenName',
319           'familyName'
320       ], $defaultFields);
321   }
```

Listing 26: ***Definition of searchable fields of an entity***

The person entity allows to search in the properties givenName and familyName.

`https://api.relator.ch/v1/api/persons?search=mike`

This URI returns a collection of `Person` resources where every person's `givenName` or `familyName` property contains the term «mike».

5 Relator GUI

The Relator GUI is implemented based on Angular in version 4.2. As the GUI is primarily a proof-of-concept prototype, I won't go much into implementation details but present the GUI as a user guide.

5.1 User Guide

The Relator GUI can be accessed by navigation to `https://app.relator.ch`. As you need to authenticate to use the application, the first screen you see is the login screen.

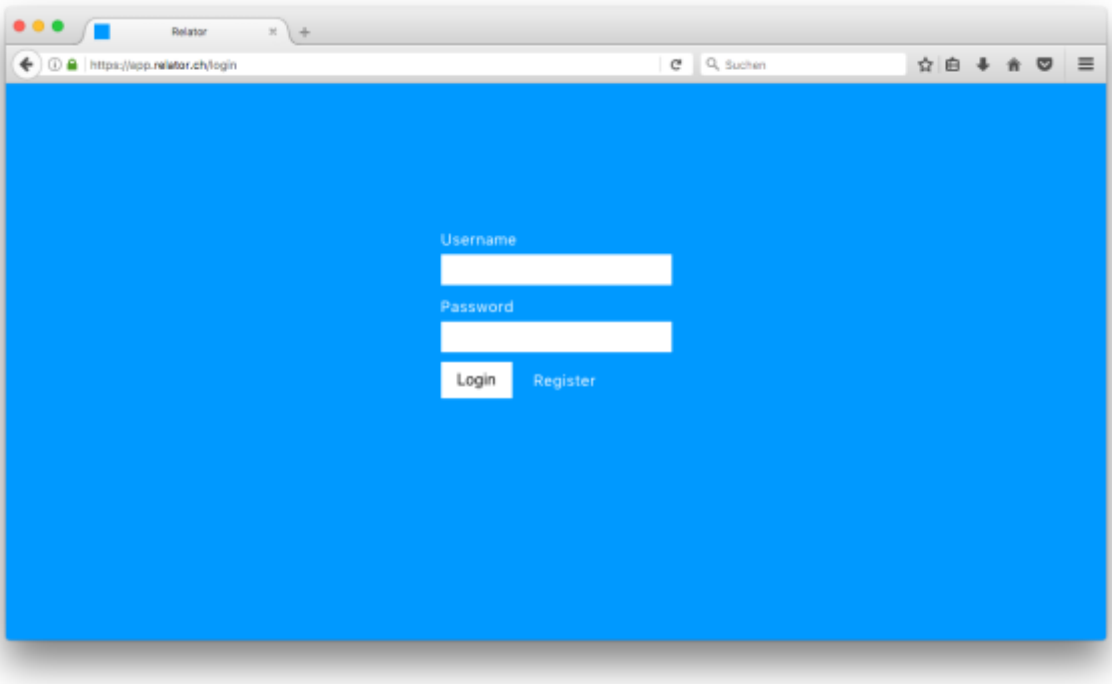


Figure 4: ***The login screen***

If you already have a user account, you can log in with your access credentials. If you don't have a user account, you can create one by clicking on the «Register» button.

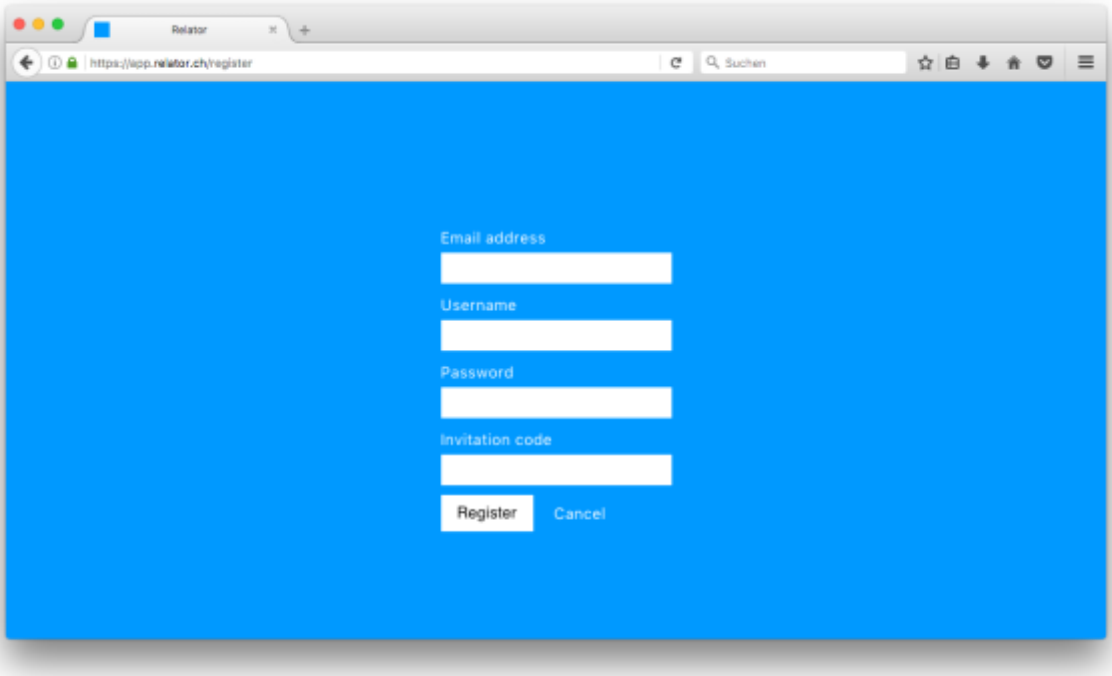


Figure 5: ***The register screen***

The register form uses both client-side validation and server-side validation. If you provide for example a wrong email address, the GUI will tell you so. If you choose a username that is already take, the GUI will tell you so, too upon clicking the «Register» button.

When you provide valid access credentials and a valid invitation code which is available on request, the register action can be completed successfully and the application will redirect you to the *dashboard*.

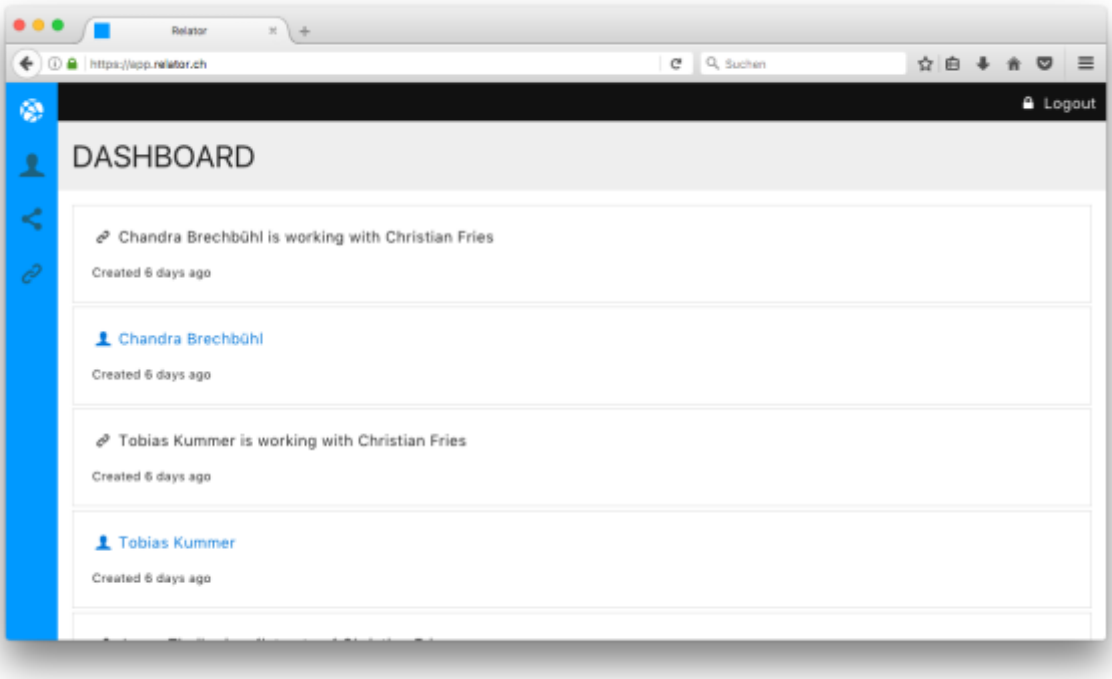


Figure 6: ***The dashboard***

As soon as you see the dashboard, you're logged in successfully. The black bar on the top is the toolbar. On the right side of the toolbar you find the logout button. The blue bar on the left is the navigation bar. It allows you to navigate between the different modules. Those elements are available on every view.

The dashboard provides an overview of the nodes that were created or modified recently. The data for the overview is fetched by sending a GET request to the endpoint **`/v1/nodes`**.

The first module in the navigation bar is the dashboard. It is active after you log in. The second module in the navigation bar is the persons module. Activate it by clicking on the icon.

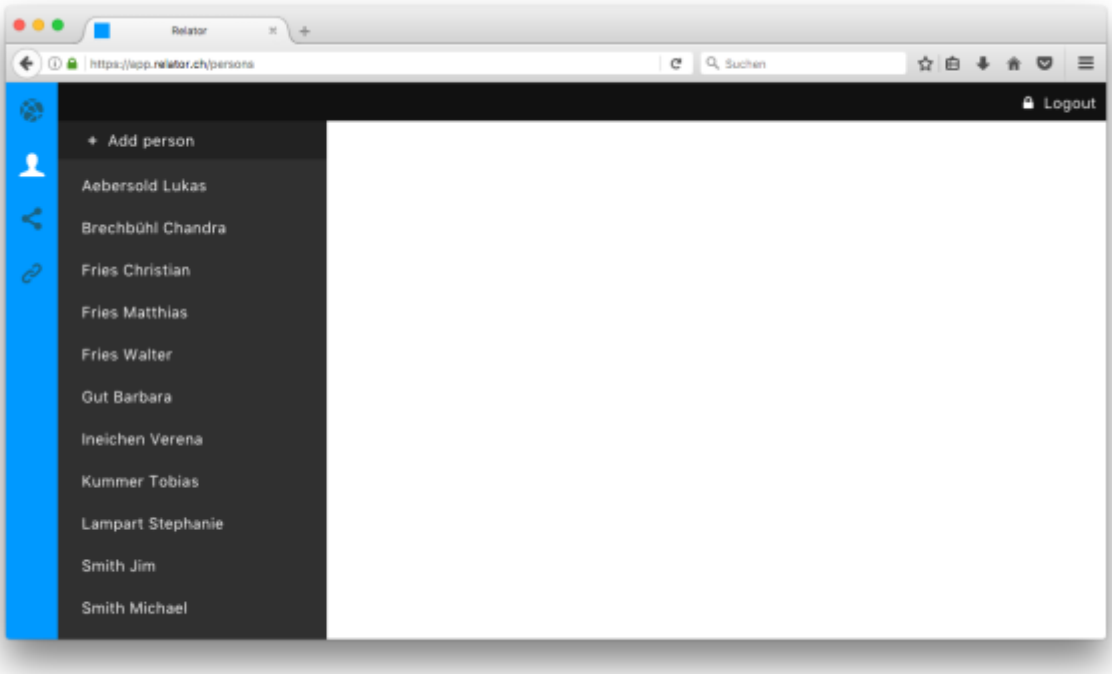


Figure 7: ***The persons module***

When you navigate to the person module, you see a list of all persons you already created on the left side next to the navigation bar. This area is called the list view. On top of that list, there's a button to create a new person.

Click on the name of a person to navigate to the detail view of that person.

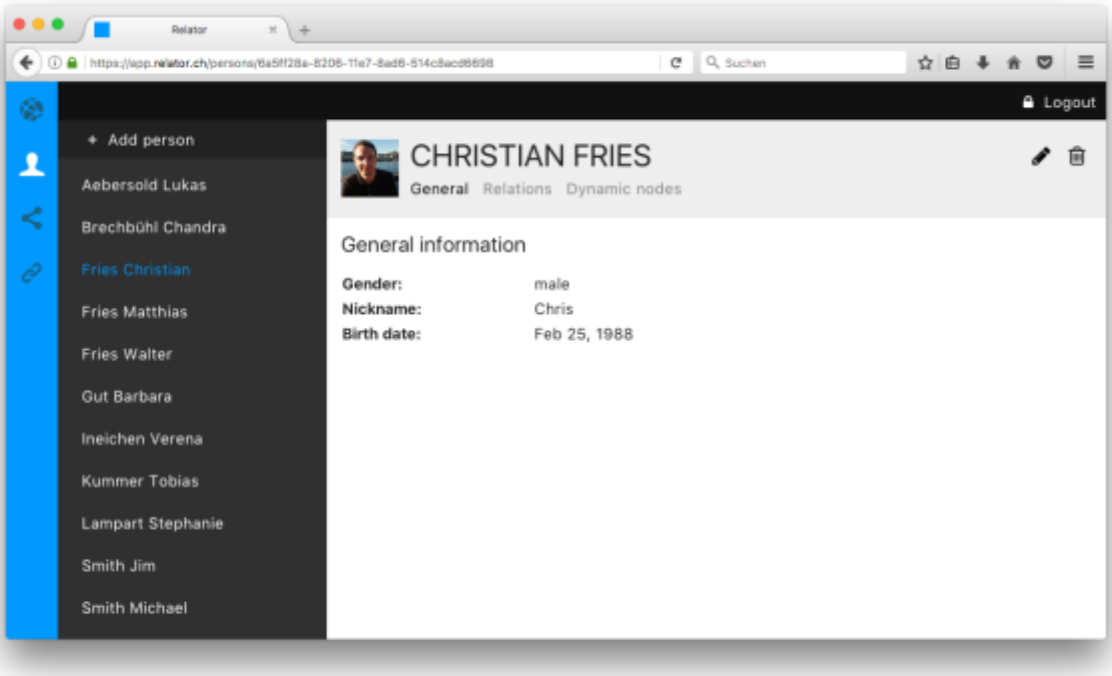


Figure 8: ***Detail view of a person***

In the content area, you can see the gray header bar with the name and the image of the person. The inline navigation below the name allows you to switch between the general information, the relations and the dynamic nodes of a person. On the right side, you find buttons to edit and to delete the person.

Below the header bar you see general information of this person.

Click the button «Add person» on top of the list view to navigate to a form to create a new person.

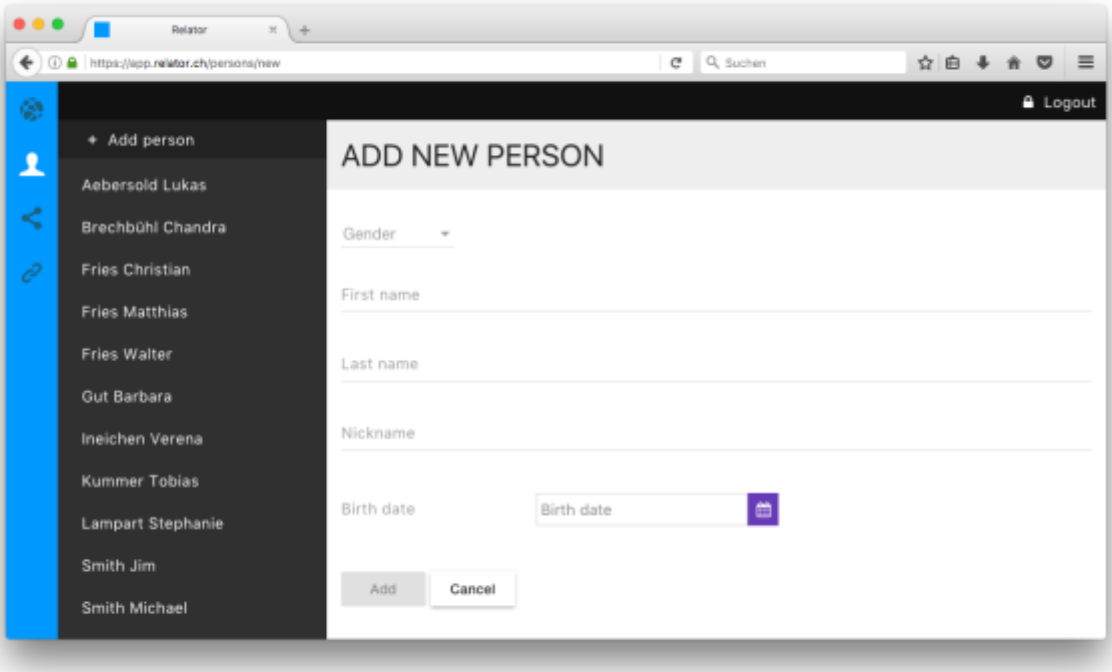


Figure 9: ***Form to add a new person***

The form fields make it very comfortable to add a new person. When you activate a required field and then activate another field without providing a value for the first field, the label and the border of the field will turn red so you see that the field is required and the value is missing.

When you load the form, the add button is disabled. It will be enabled once all required form fields are properly filled. When you hit the «Add» button, the new person will be created and you'll see the detail view of that newly created person.

Let's have a look at the third module, the dynamic nodes module. Navigate to that module by clicking the third icon in the navigation bar.

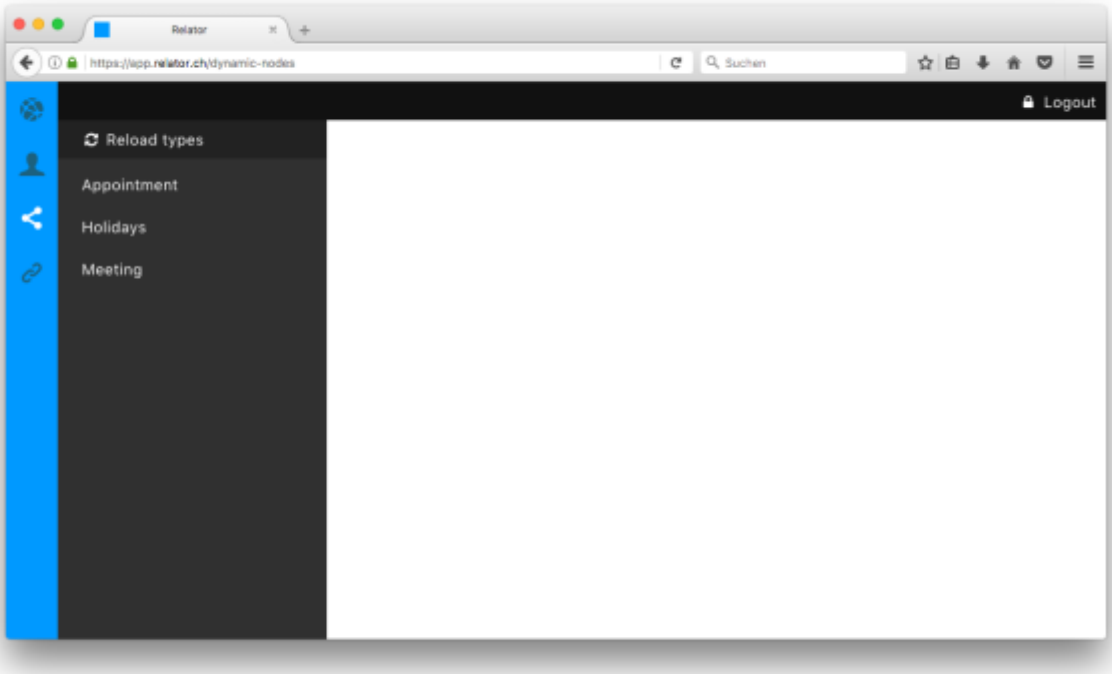


Figure 10: ***The dynamic nodes module***

This module presents all existing dynamic node types in the list view. The button on top of the list view reloads the list. This is useful because the application does not provide a form to create and edit dynamic node types. You can find out how to create and edit them by using the API directly as explained in the next section. When you create or edit a dynamic node type externally, hit the «Reload types» button to refresh the list.

When you click on a dynamic node type in the list view, the dynamic nodes of the selected type are shown in the content area.

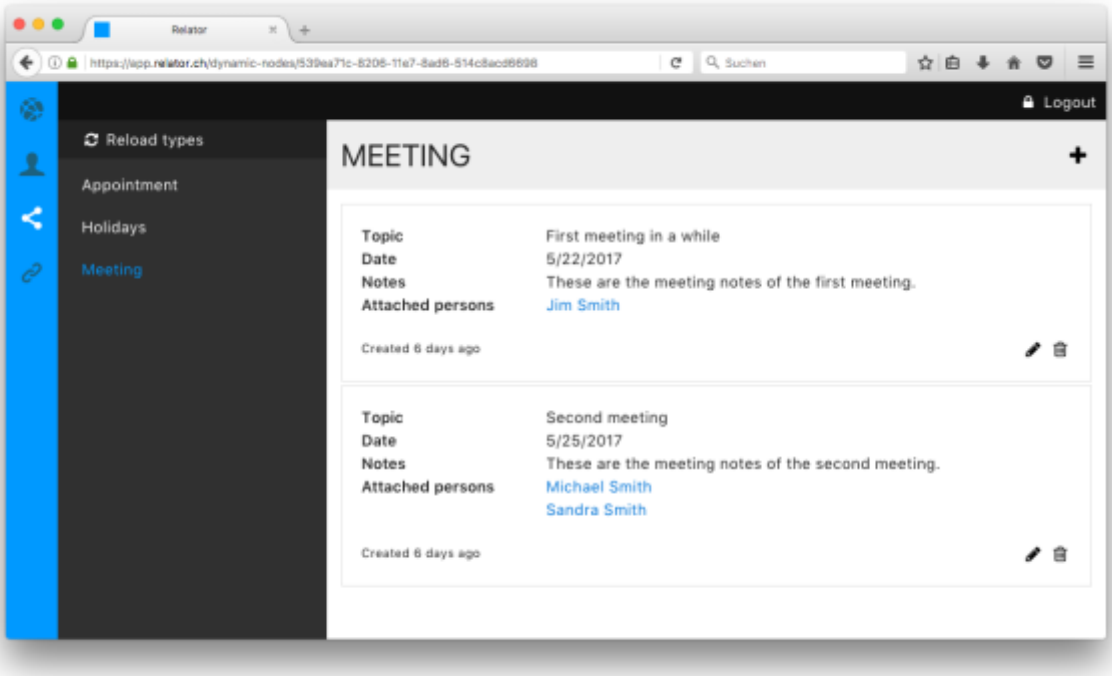


Figure 11: ***Dynamic nodes of a selected dynamic node type***

The header bar shows the title of the dynamic node type selected and a button to create a new dynamic node based on this type. The content are lists the dynamic nodes of the selected type in descending order. The newest dynamic node is always on top. Each dynamic node has buttons to edit or delete it.

A click on the plus button navigates to a form to create a new dynamic node.

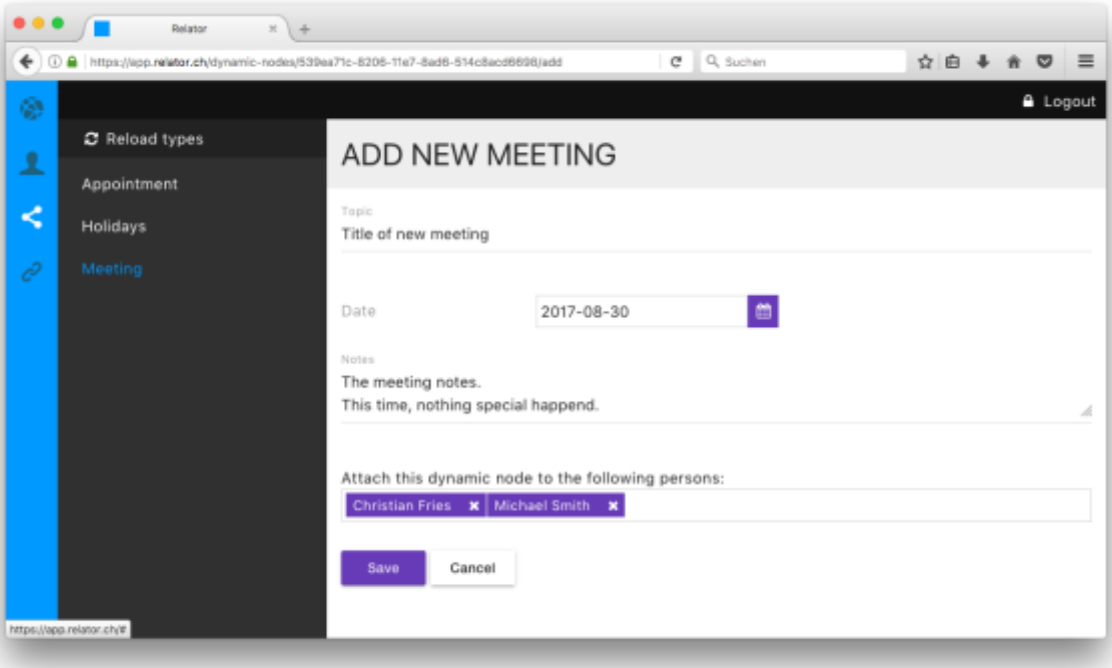


Figure 12: ***Form to add a new dynamic node***

The form fields you see in Figure 12 are dynamically generated based on the `configuration` property of the assigned dynamic node type.

The last field allows you to attach the dynamic node to multiple persons. Activate the field and start typing a name. The field will suggest you persons based on your input.

Let's have a look at the fourth module, the relation types module. Navigate to that module by clicking the fourth icon.

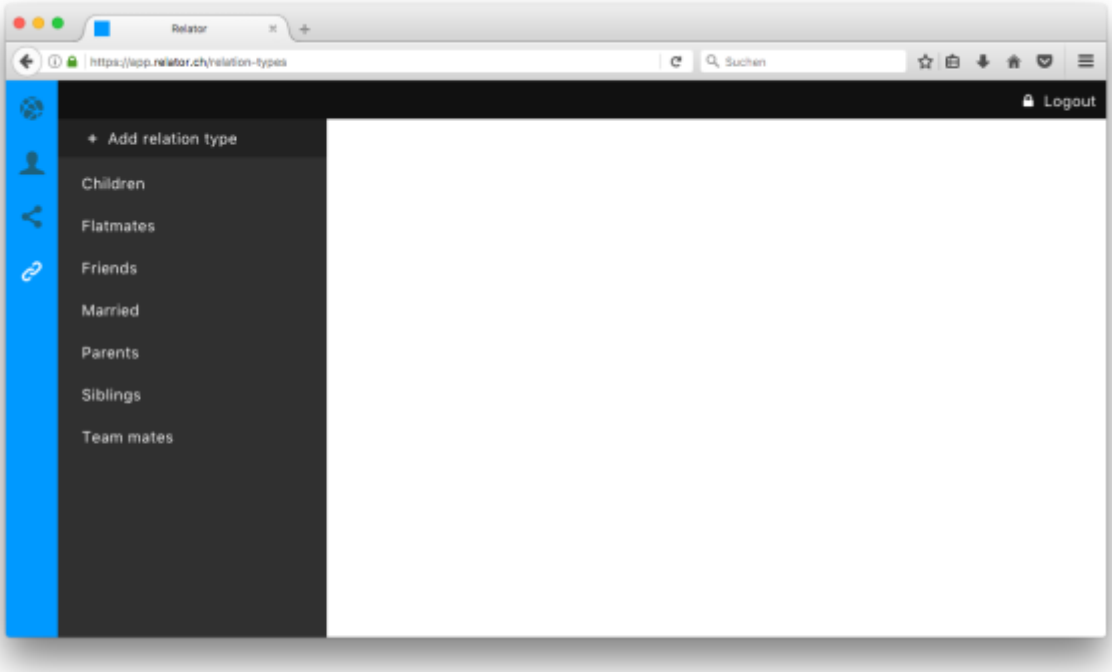


Figure 13: ***The relations types module***

This module allows you to create and edit relation types. You can assign these relation types in the persons module later.

The list view lists all existing relation types. The button on top of that list allows you to create new relation types.

Click on the name of a relation type to navigate to the detail view of a relation type.

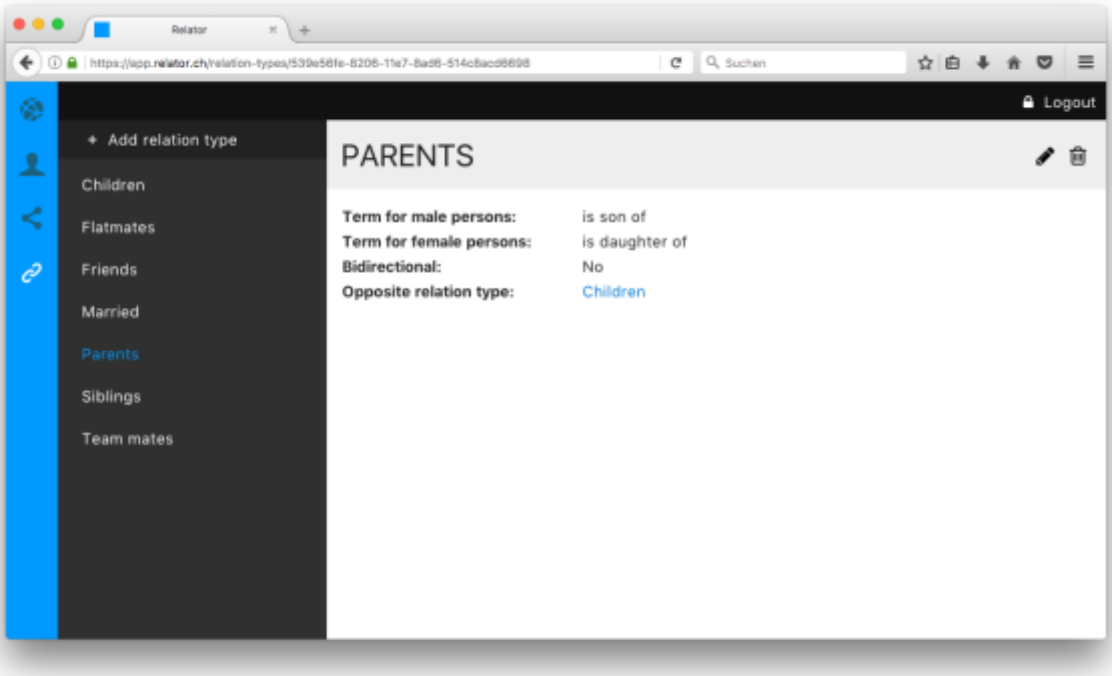


Figure 14: ***Detail view of a relation type***

In the header bar, you see the name of the selected relation type and the buttons to edit and delete it. In the content area, you see the configuration of the relation type. If a relation type is non-bidirectional, a link to the opposite relation type is presented.

Create, edit or remove relation types in this module according to your needs. Please mind that when you remove a relation type, all relations based on that type will be removed as well.

Once you created the needed relation types, navigate to the person module, select the person you want to add a relation to and choose «Relations» from the inline navigation.

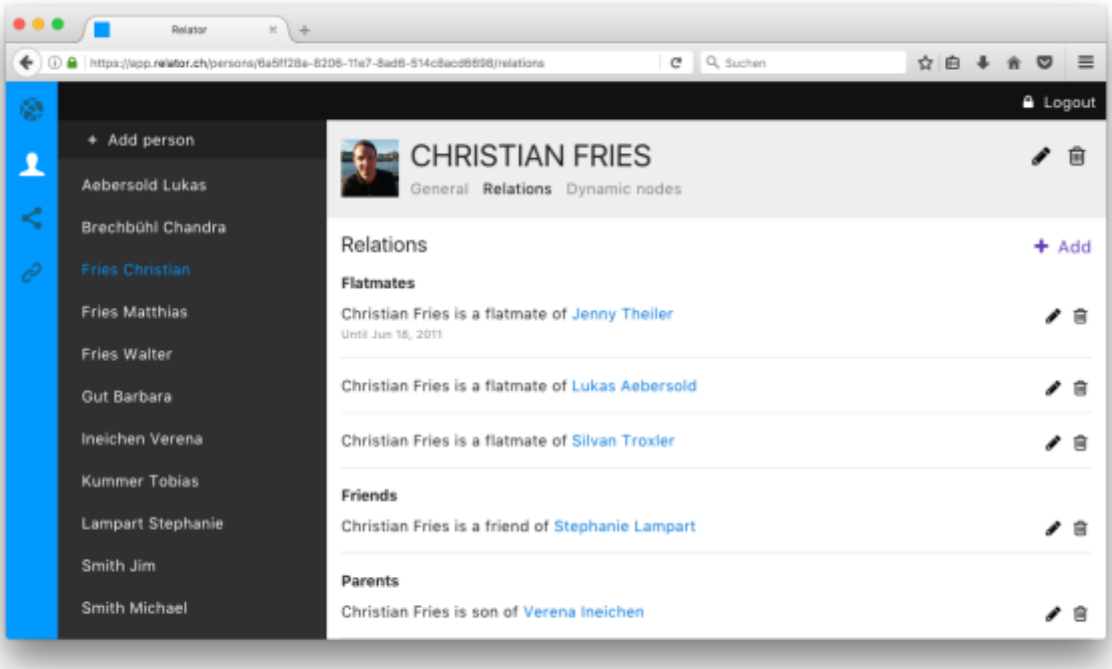


Figure 15: ***List of relations of a person***

The content area presents all relations that were already created grouped by the relation type. Every relation offers buttons to edit or remove the relation.

Click the «Add» button in the top right corner of the content area to add a new relation.

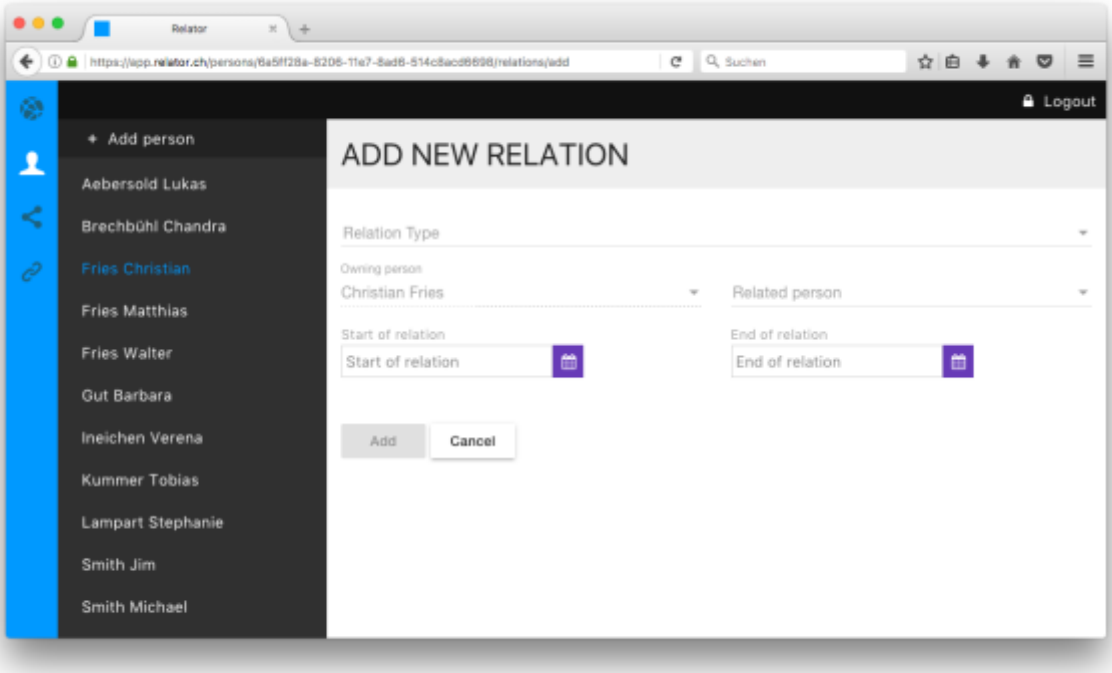


Figure 16: ***Add new relation to a person***

The form to create a new relation lets you choose a relation type, the related person and optionally, the start and the end date of the relation. The owning person is already preselected, it's the person whose detail view was active when you clicked the «Add» button.

If you want to see what dynamic nodes are attached to a certain person, navigate to the detail view of that person and choose «Dynamic nodes» from the inline navigation.

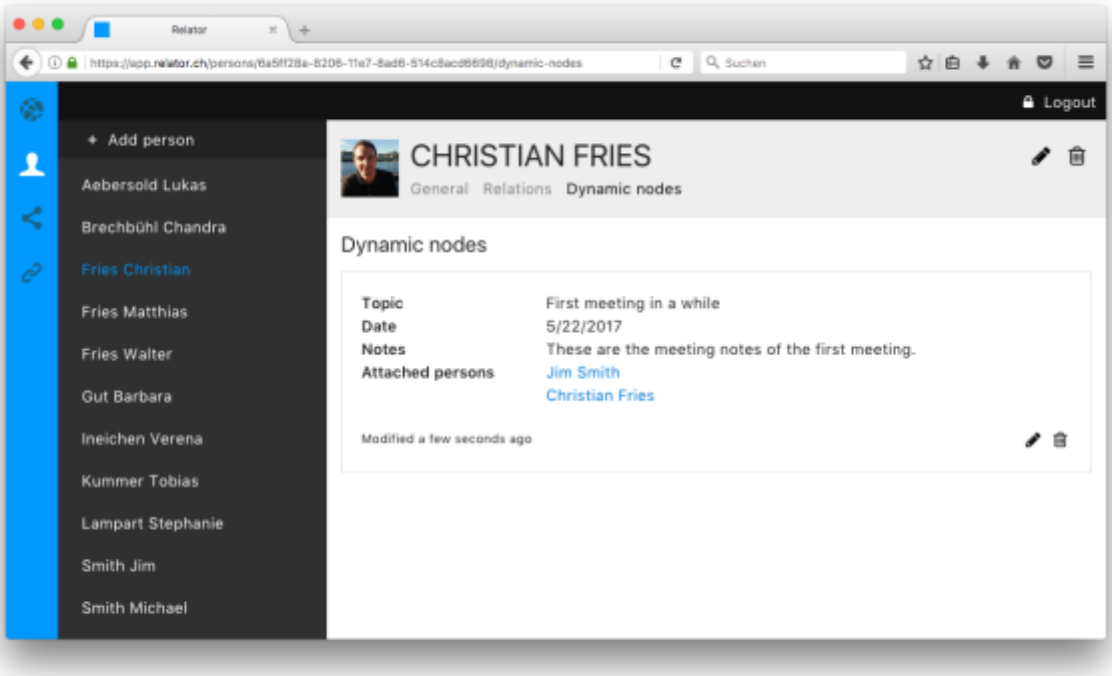


Figure 17: ***List of dynamic nodes attached to a selected person***

This view lists all dynamic nodes that are attached to the currently selected person. Every dynamic node has buttons to edit and remove that node.

5.2 Advanced Options

The Relator GUI doesn't provide an interface to create dynamic node types. To create and edit dynamic node types, a client has to access the API directly. Also, all actions that can be performed by using the Relator GUI can be performed by accessing the API directly.

A simple tool to interact with REST APIs is *Postman* [13]. Postman is a cross-platform tool available for free as a standalone version or as an extension for the browser Chrome. It makes it very easy to interact with APIs even if they require authentication.

How to use Postman and the Relator API

After setting up the application, Postman asks you to create an account. This account is very handy as it stores your recent requests and you can create collections of requests that are synchronized to all of your devices.

Once your account is set up, you are ready to use Postman.

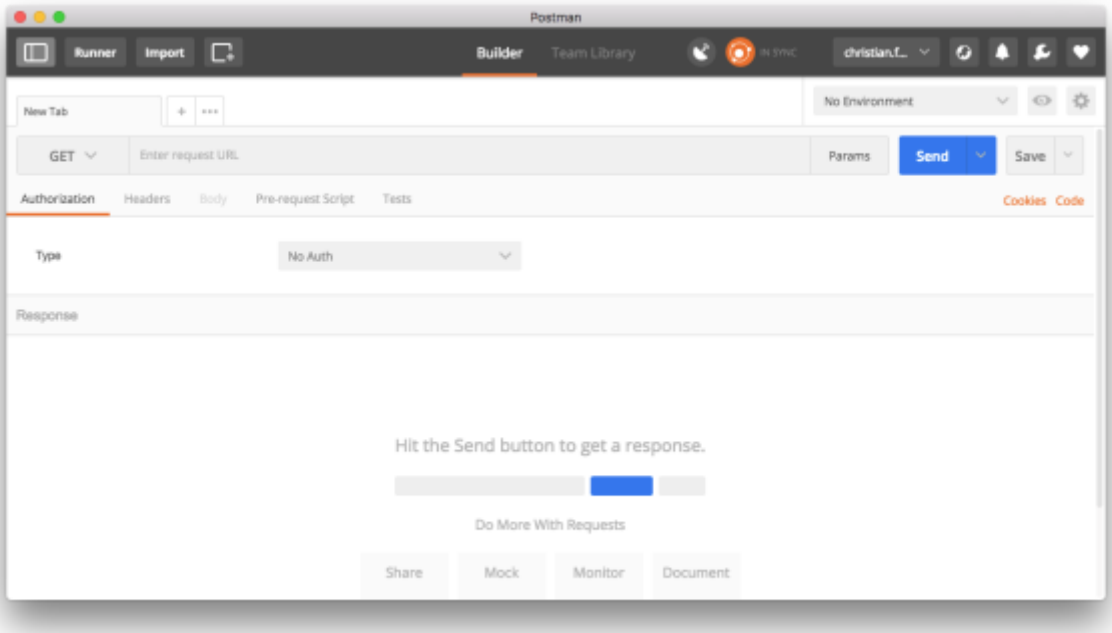


Figure 18: ***The Postman UI***

The first thing you have to do when you want to access the Relator API is to create an access token. This can be achieved by sending a `POST` request to the endpoint **`/v1/user/token`**.

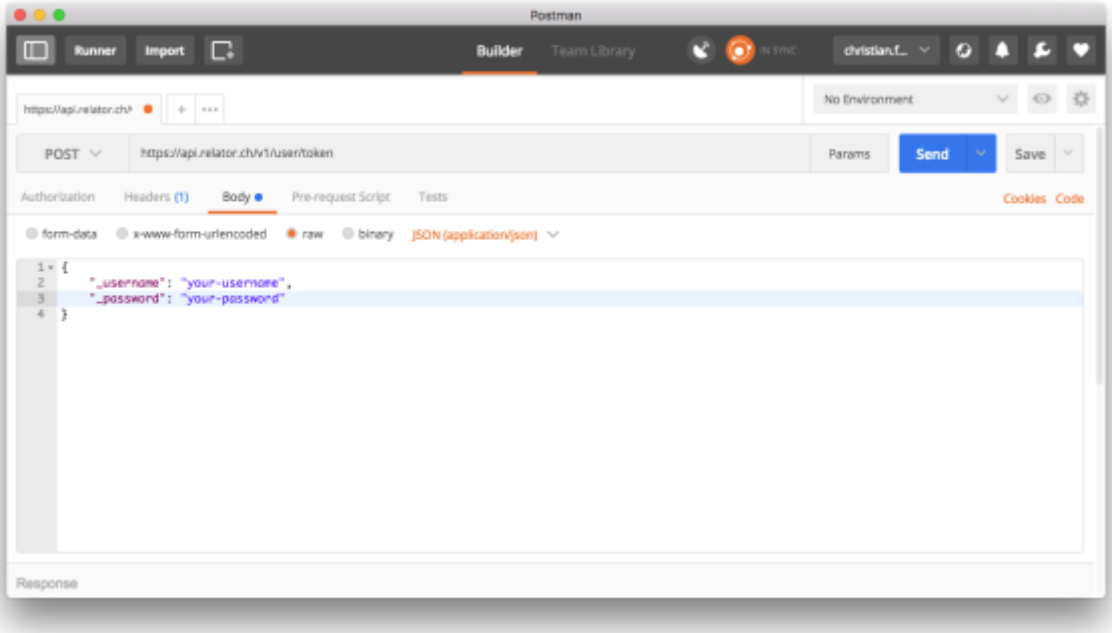


Figure 19: ***Authentication with Postman***

Setup Postman as shown in Figure 19. When you hit the «Send» button, Postman will send the request to the Relator API. If your access credentials are wrong, the response will have a status code of `400` or `401` and the response body will contain an error message.

If your access credentials are correct, the response body will contain a property `token` with a value of a long cryptic string. That string is your access token.

Copy the access token and open a new tab. Setup the tab as shown in Figure 20.

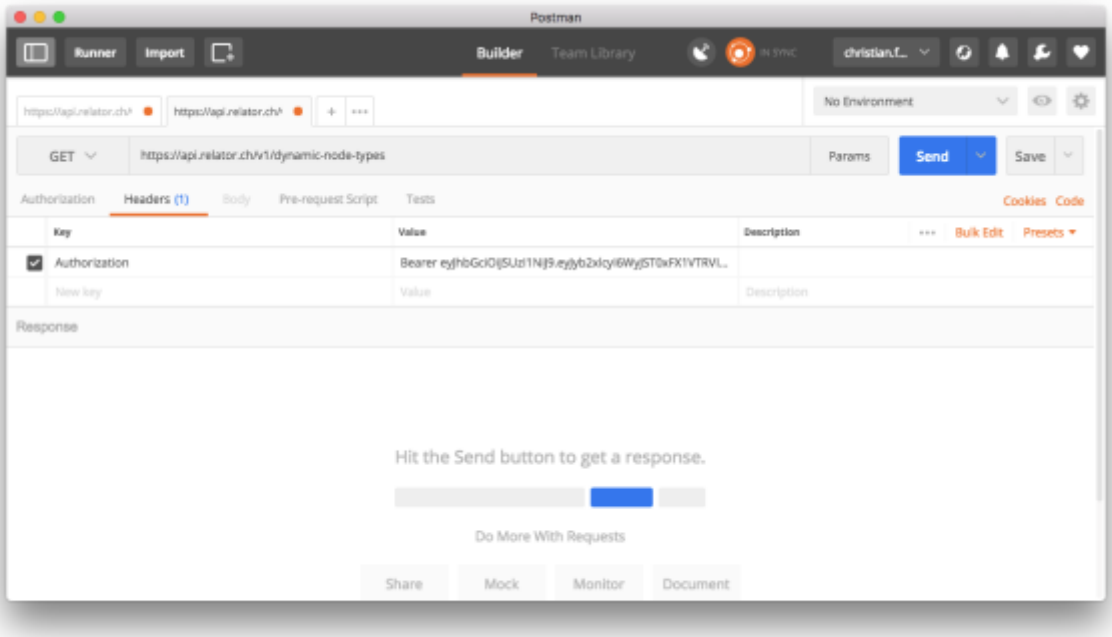


Figure 20: ***Accessing a protected endpoint with Postman***

The most important part is that you set the `Authorization` header in the `Headers` tab. Add a key called `Authorization` and for the value, type «Bearer», add a space and paste in the access token. This is required to access protected endpoints. When you hit the «Send» button, your response will contain a collection of dynamic node types. If you didn't create any so far, the collection will be empty.

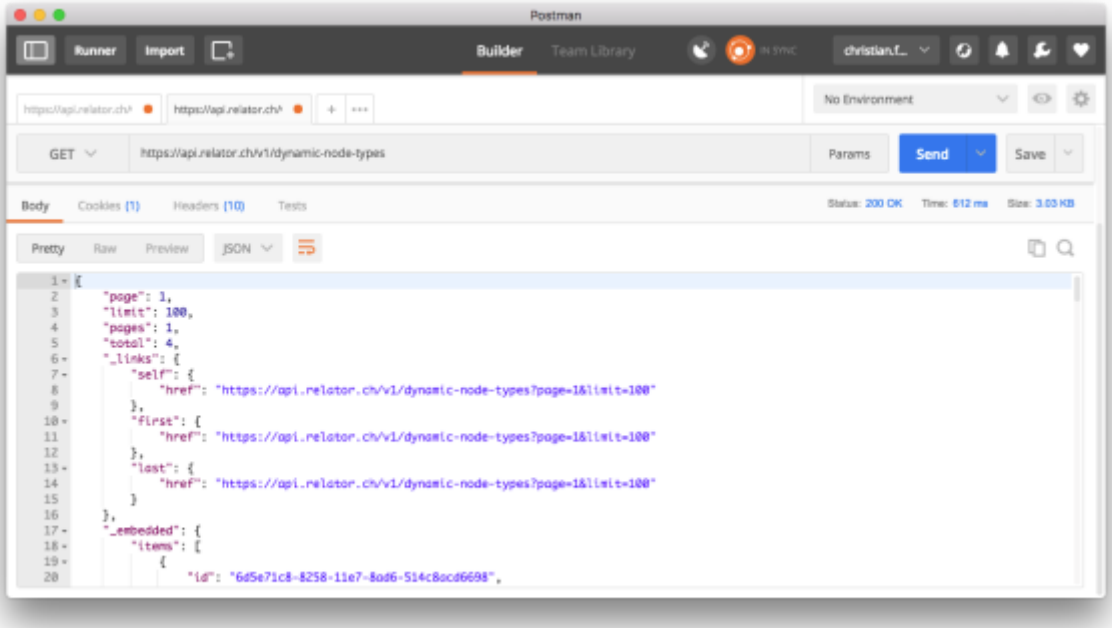


Figure 21: ***Collection of dynamic node types after successful authentication***

Now you can use all the endpoints of the API described in the Swagger documentation.

6 Conclusion

6.1 Lessons Learned

One of the biggest lessons learned in this project is the fact that choosing the right tools to create an application is a very important task. I chose to use a relational database system to store my application data because I already had knowledge in how to work with this type of databases. Later on, during the project, I was reading about graph database systems and found out that it could have been a better choice to achieve the goals of my project. I cannot say it would have been better for sure as I did not try it out but for me, doing research to find the right tools and having courage to try something new is a lesson I learned.

Another lesson I learned: Think about the exact use case for your API before you start working on it. Developer's like to create universal things that can handle all kind of use cases and so do I. But once I started implementing the Relator GUI, I found out that I cannot handle certain requirements of the GUI because the API was created in a too general and too optimized way. As an example, certain properties of the Relation entity were only exposed if the Relation was requested directly but not when it was part of a collection. The idea was saving bandwidth by not exposing these properties but during implementation of the Relator GUI I found out that they were necessary.

6.2 Future Improvements

As the Relator GUI is only a prototype and does not offer a UI to manage dynamic node types and groups, one of the first improvements would be the GUI. Besides the missing UI parts, the user experience could be improved by adding context sensitive help and better validation handling.

The big deal would be to improve the UI to be completely responsive so that it can be used on smartphones. Currently it's only possible to use the UI on tablets and on desktop devices.

Concerning the API, the next important step would be to add more field types to the dynamic nodes. Types like email, phone, link, location etc. would make the dynamic nodes much more powerful.

Another important improvement would be adding more fields to the `Person` entity like addresses, phone numbers and email addresses. If the GUI would be completely responsive, the Relator application could be used as an organizer.

6.3 Final Statement

In the introduction of this project I mentioned the importance of the world wide web and the huge impact it has on our daily lives. After doing some research about architecture styles used for web services, I found out that REST is very popular and that most of the services I use in my daily live offer a RESTful API. Based on that research I'm sure that the REST architecture style helped a lot in the evolvement of the internet and internet-based services in the recent years and I am convinced that it will continue helping the internet to grow, to evolve and to provide a solid and modern base for new applications and services.

A
Project Files

The project files of this project are available here:
https://resources.relator.ch/project_files.zip

Directory structure

relator_api

This directory contains the source code for the Relator API.

relator_gui

This directory contains the source code for the Relator GUI.

resources

This directory contains resources like images used in this report.

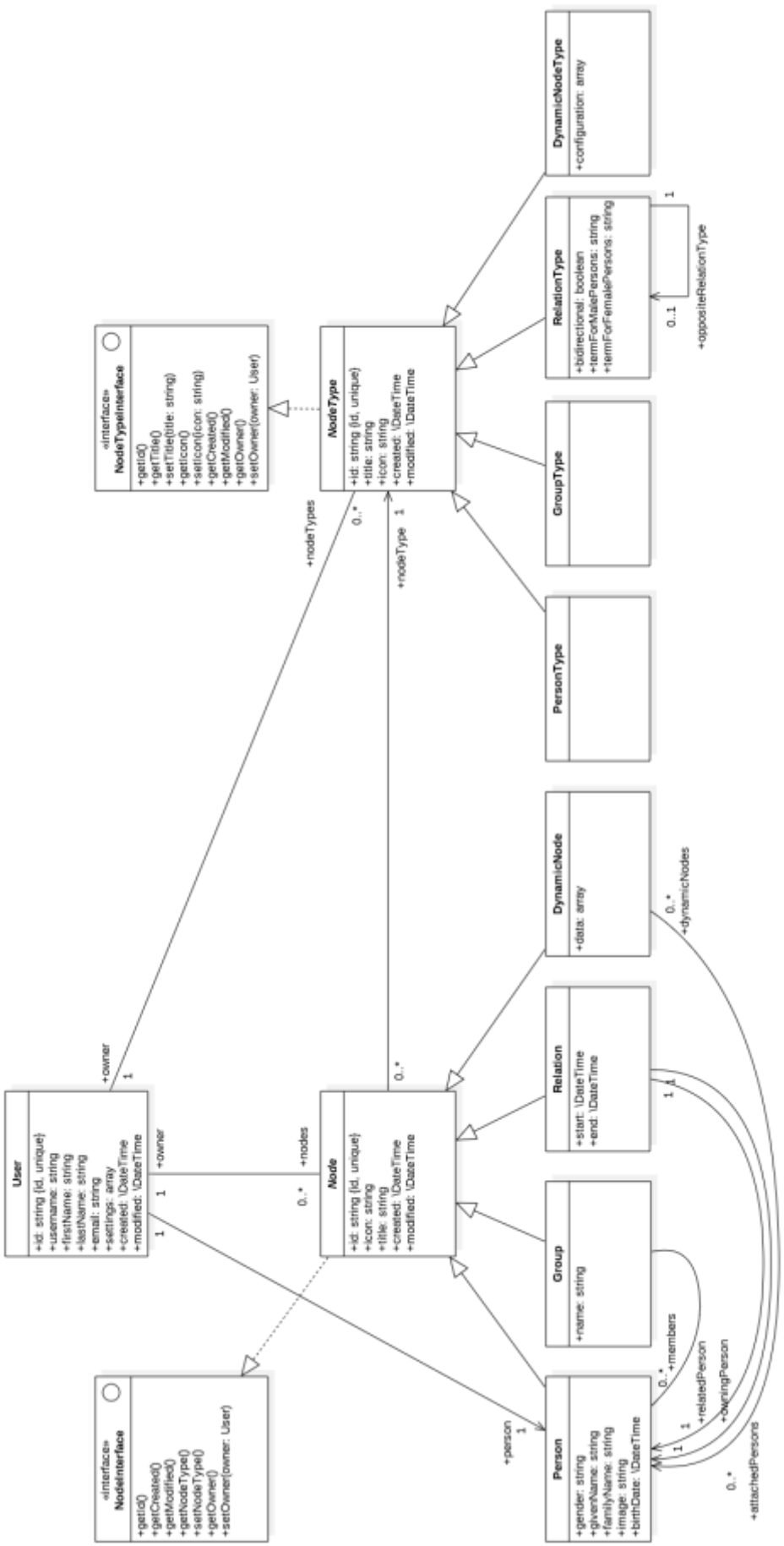
static_api_documentation

This directory contains the static html version of the Swagger API documentation.

B

Domain Model

The schema of the full Relator domain model can be found on the next page.


«Interface»
NodeTypeInterface
+getId()
+getTitle()
+setTitle(title: string)
+getIcon()
+setIcon(icon: string)
+getCreated()
+getModified()
+getOwner()
+setOwner(owner: User)
NodeType
+id: string (Id, unique)
+title: string
+icon: string
+created: \DateTime
+modified: \DateTime
DynamicNodeType
+configuration: array
RelationType
+bidirectional: boolean
+termForMalePersons: string
+termForFemalePersons: string
+oppositeRelationType
GroupType
PersonType
+nodeTypes
+nodeType
0..*
1
User
+id: string (Id, unique)
+username: string
+firstName: string
+lastName: string
+email: string
+settings: array
+created: \DateTime
+modified: \DateTime
+owner
+nodes
Node
+id: string (Id, unique)
+icon: string
+title: string
+created: \DateTime
+modified: \DateTime
DynamicNode
+data: array
Relation
+start: \DateTime
+end: \DateTime
+dynamicNodes
Group
+name: string
Person
+gender: string
+givenName: string
+familyName: string
+image: string
+birthDate: \DateTime
+members
+relatedPerson
+owningPerson
+attachedPersons
+person
«Interface»
NodeInterface
+getId()
+getCreated()
+getModified()
+getNodeType()
+setNodeType()
+getOwner()
+setOwner(owner: User)

References

[Fie00]

Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*, University of California, Irvine, 2000. [Retrieved August 03, 2017 from http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf]

[Bur13]

Bill Burke. *RESTful Java with JAX-RS 2.0*, O'Reilly Media, 2013.

[Ber05]

Berners-Lee, et al. *Uniform Resource Identifier (URI): Generic Syntax*, Network Working Group, 2005. [Retrieved August 04, 2017 from https://tools.ietf.org/pdf/rfc3986.pdf]

[FR14]

Roy Thomas Fielding and Julian Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*, IETF, 2014. [Retrieved August 03, 2017 from https://tools.ietf.org/pdf/rfc7231.pdf]

[Lea05]

Leach, et al. *A Universally Unique Identifier (UUID) URN Namespace*, Network Working Group, 2005. [Retrieved August 11, 2017 from https://tools.ietf.org/pdf/rfc4122.pdf]

Referenced Web Resources

[1] Symfony, High Performance PHP Framework for Web Development. http://symfony.com (accessed October 12, 2016)

[2] PHPUnit. https://phpunit.de/ (accessed December 03, 2016)

[3] Friends of Symfony: User Bundle. https://github.com/FriendsOfSymfony/FOSUserBundle (accessed on November 29, 2016).

[4] Doctrine: Object-relational Mapper. http://www.doctrine-project.org/projects/orm.html (accessed on November 29, 2016)

[5] Bitbucket. https://bitbucket.org/ (accessed on October 19, 2016)

[6] CircleCI. https://circleci.com/ (accessed on October 20, 2016)

[7] Capistrano. http://capistranorb.com/ (accessed on October 23, 2016)

[8] Swagger. https://swagger.io/ (accessed November 29, 2016)

[9] Doctrine: Inheritance mapping. http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/inheritance-mapping.html (accessed on November 30, 2016)

[10] JSON Web Tokens. https://jwt.io/ (accessed on December 08, 2016)

[11] JMSSerializerBundle. https://jmsyst.com/bundles/JMSSerializerBundle (accessed on November 12, 2016)

[12] BazingaHateoasBundle. https://github.com/willdurand/BazingaHateoasBundle (accessed on January 12, 2017)

[13] Postman. https://www.getpostman.com/ (accessed November 12, 2016)