Frogr Documentation

Release 0.2.3-SNAPSHOT

Jochen Weis

Oct 26, 2018

Contents:

Frogr is a Java framework for developing high-performance RESTful web services.

With **Frogr** you can get a service up and running in minutes, but there's no limit in complexity. **Frogr** uses a *repository pattern* for seperating data, business logic and external interfaces. This approach makes it easy to test and extend your code.

Frogr packs multiple stable libraries into an easy to use, light-weight framework, so you won't have tons of additional dependencies, you probably never will need.

Java:

```
PersonRepository repository = service().repository(Person.class);
List<Person> persons = personRepository.search()
  .query("*Smith"))
  .fields("name", "children.name")
  .list()
```

REST:

```
GET localhost:8282/persons?q=*Smith&fields=name,children.name
```

```
{
  "success": true,
  "data": [
    {
      "uuid": "99eaeddc012811e883fda165f97f8411",
      "type": "Person",
      "name": "Beth Smith",
      "children": [
        {
          "uuid": "99eb630e012811e883fd97343769a63e",
          "type": "Person",
          "name": "Morty Smith"
        },
        {
          "uuid": "99ebb12f012811e883fd2583ad59c919",
          "type": "Person",
          "name": "Summer Smith"
        }
      ]
    },
    {
      "uuid": "99eb3bfd012811e883fdd551cdefd5ed",
      "type": "Person",
      "name": "Jerry Smith",
      "children": [
        {
          "uuid": "99eb630e012811e883fd97343769a63e",
          "type": "Person",
          "name": "Morty Smith"
        },
        {
          "uuid": "99ebb12f012811e883fd2583ad59c919",
          "type": "Person",
          "name": "Summer Smith"
        }
      ]
    },
```

(continues on next page)

(continued from previous page)

```
    {
      "uuid": "99eb630e012811e883fd97343769a63e",
      "type": "Person",
      "name": "Morty Smith",
      "children": []
    },
    {
      "uuid": "99ebb12f012811e883fd2583ad59c919",
      "type": "Person",
      "name": "Summer Smith",
      "children": []
    }
  ]
}
```

CHAPTER 1

Quickstart

First, add the dependency to your project.

1.1 Maven

```
<dependency>
    <groupId>de.whitefrog</groupId>
    <artifactId>frogr-base</artifactId>
    <version>0.2.3-SNAPSHOT</version>
</dependency>
```

1.2 Application

Next we will create the main entry point for the service.

```
public class MyApplication extends Application<Configuration> {
  private MyApplication() {
    // register the rest classes
    register("de.whitefrog.frogr.example.basic.rest");
    // register repositories and models
    serviceInjector().service().register("de.whitefrog.frogr.example.basic");
    // use common config instead of the default config/neo4j.properties
    serviceInjector().service().setConfig("../config/neo4j.properties");
  }

  @Override
  public String getName() {
    return "frogr-base-example";
  }

```

(continues on next page)

(continued from previous page)

```
  public static void main(String[] args) throws Exception {
    new MyApplication().run("server", "../config/example.yml");
  }
}
```

As you can see there are two registry calls in the application's constructor. `register(...)` let's the application know in which package to look for rest classes. `serviceInjector().service().register(...)` tells the application where to look for models and repositories. More information about the Application entry point: *Application*

1.3 Configs

You may also have noticed there's a config file used in the main method. This is required to setup our Dropwizard instance, so we have to create that one now. There's a second config file needed, which configures our embedded Neo4j instance. By default these configs should be in your project in a directory 'config'.

`config/example.yml`

```
server:
    applicationConnectors:
      - type: http
        port: 8282
    adminConnectors:
      - type: http
        port: 8286
logging:
    level: WARN
    loggers:
        de.whitefrog.frogr: INFO
        io.dropwizard.jersey.DropwizardResourceConfig: INFO
        io.dropwizard.jersey.jackson.JsonProcessingExceptionMapper: DEBUG
    appenders:
      # console logging
      - type: console
        logFormat: '[%d] [%-5level] %logger{36} - %msg%n'
```

Reference: Dropwizard Configuration

`config/neo4j.properties`

```
graph.location=target/graph.db
```

This file is not required, by default the graph.location is "graph.db" inside your working directory. Reference: Neo4j Configuration

1.4 RelationshipTypes

We should add a class that holds our relationship types, so that we have consistent and convienient access. This is not a requirement but I highly recommend it. Doing so we don't have to deal with strings in Java code, which is never a good choice, right?

```
object RelationshipTypes {
  const val ChildOf = "ChildOf"
  const val MarriedWith = "MarriedWith"
}
```

1.5 Model

Now, let's create a *model*. I recommend using Kotlin for that. All models have to extend the Entity class or implement the Model interface at least.

```
class Person() : Entity() {
  constructor(name: String) : this() {
    this.name = name
  }

  // Unique and required property
  @Unique
  @Indexed(type = IndexType.LowerCase)
  @Required
  var name: String? = null

  // Relationship to another single model
  @RelatedTo(type = RelationshipTypes.MarriedWith, direction = Direction.BOTH)
  var marriedWith: MarriedWith? = null
  // Relationship to a collection of models
  @Lazy
  @RelatedTo(type = RelationshipTypes.ChildOf, direction = Direction.OUTGOING)
  var parents: List<Person> = ArrayList()
  @Lazy
  @RelatedTo(type = RelationshipTypes.ChildOf, direction = Direction.INCOMING)
  var children: List<Person> = ArrayList()

  companion object {
    @JvmField val Name = "name"
    @JvmField val MarriedWith = "marriedWith"
    @JvmField val Children = "children"
  }
}
```

As you can see, we used the relationship types created before, to declare our relationships to other models.

1.6 Repository

Normally we would create a repository for persons. But we won't need extra methods for this tutorial and frogr will create a default repository if it can't find one. If you need more information visit *Repositories*.

1.7 Service

Next we'll have to create the REST *service* layer. There's a base class, that provides basic CRUD operations, so you only have to add methods for special cases. Of course you can also use any other JAX-RS annotated class.

```
@Path("person")
public class Persons extends CRUDService<PersonRepository, Person> {
  @GET
  @Path("init")
  public void init() {
    // insert some data
    try(Transaction tx = service().beginTx()) {
      if(repository().search().count() == 0) {
        repository().init();
        tx.success();
      }
    }
  }
}
```

1.8 Examples

You can find the code used in this guide and more examples at Github

CHAPTER 2

Documentation

2.1 Application

In our main `Application` class, we can implement additional features, such as a custom `Service` implementation, metrics, or authorization. See the chapter on *Authorization and Security* for further information.

```java
public class MyApplication extends Application<Configuration> {
  private MyServiceInjector serviceInjector;

  private MyApplication() {
    // register the rest classes
    register("de.whitefrog.frogr.example.customservice.rest");
    // register repositories and models
    serviceInjector().service().register("de.whitefrog.frogr.example.customservice");
  }

  // override to pass our own ServiceInjector implementation
  @Override
  public ServiceInjector serviceInjector() {
    if(serviceInjector == null) {
      serviceInjector = new MyServiceInjector();
    }
    return serviceInjector;
  }

  @Override
  public void run(Configuration configuration, Environment environment) throws␣
↪Exception {
    super.run(configuration, environment);

    // bind the custom ServiceInjector to our Service implementation, described below
    environment.jersey().register(new AbstractBinder() {
      @Override
      protected void configure() {
```

(continues on next page)

(continued from previous page)

```
        bindFactory(serviceInjector()).to(MyService.class);
      }
    });

    // register metrics
    environment.jersey().register(new␣
↪InstrumentedResourceMethodApplicationListener(RestService.metrics));

    // add a console reporter for the metrics
    final ConsoleReporter reporter = ConsoleReporter.forRegistry(RestService.metrics)
      .convertRatesTo(TimeUnit.SECONDS)
      .convertDurationsTo(TimeUnit.MILLISECONDS)
      .build();
    // report every 30 minutes
    reporter.start(30, TimeUnit.MINUTES);

    // add a logger reporter for the metrics
    final Slf4jReporter slf4j = Slf4jReporter.forRegistry(RestService.metrics)
      .outputTo(LoggerFactory.getLogger("com.example.metrics"))
      .convertRatesTo(TimeUnit.SECONDS)
      .convertDurationsTo(TimeUnit.MILLISECONDS)
      .build();
    // report every 30 minutes
    slf4j.start(30, TimeUnit.MINUTES);
  }

  @Override
  public String getName() {
    return "example-rest";
  }

  public static void main(String[] args) throws Exception {
    new MyApplication().run("server", "config/example.yml");
  }
}
```

2.1.1 Service Injector

We can also write our own `ServiceInjector`, in case we want to override the base `Service`.

```
public class MyServiceInjector extends ServiceInjector {
  private MyService service;

  public MyServiceInjector() {
    service = new MyService();
  }

  @Override
  public Service service() {
    return service;
  }

  @Override
  public Service provide() {
    if(!service.isConnected()) service.connect();
```

(continues on next page)

(continued from previous page)

```
    return service;
  }

  @Override
  public void dispose(de.whitefrog.frogr.Service service) {
    service.shutdown();
  }
}
```

In that case, `Service` is our own implementation and should extend `de.whitefrog.frogr.Service`.

2.1.2 Service

For instance if we want to provide a common configuration accessible from any *Repository* or *Service*:

```
public class MyService extends Service {
  private static final Logger logger = LoggerFactory.getLogger(MyService.class);
  private static final String appConfig = "config/myapp.properties";

  private Configuration config;

  @Override
  public void connect() {
    try {
      config = new PropertiesConfiguration(appConfig);
      super.connect();
    } catch(ConfigurationException e) {
      logger.error(e.getMessage(), e);
      throw new RuntimeException(e);
    }
  }

  public Configuration config() { return config; }
}
```

2.2 Models

Models define the data structure, used for our project. Each model has at least extend the `Model` interface.

2.2.1 Entities

For entities there's a base class `Entity` that implements all needed methods for us.

Entity models primarily consists of field and relationship definitions that define how it is used inside our project. I recommend using Kotlin for models, because I'm lazy and hate to write getter and setter methods, but thats up to you ;)

2.2.2 Relationships

For relationships we can use the `BaseRelationship<From, To>` class.

We can use every annotated field entities are using, except `@RelatedTo` and `@RelationshipCount`.

2.2.3 Fields

Model fields should always be initialized with `null`, so that the persistence layer can properly decide if the value is relevant for storing to database. Also we should not use primitive types here.

Fields can be annotated with hibernate annotations extended by a set of unique ones. These are:

`@Fetch` Indicates that a field should be automatically fetched.

`@Indexed(type=IndexType.Default)` Indicates that a field should be handled by an index.

`type` can be either IndexType.Default or IndexType.LowerCase for case-insensitive matches.

`@Lazy` Indicator for lists to fetch them lazily on demand, not every list item at once. Using this will NOT delete relationships when one is missing in save operations. We have to delete them manually, when needed.

`@NotPersistant` The field should not be persisted.

`@NullRemove` Remove a property if set to `null`. Note that this will also delete the property if we save a model where the @NullRemove property is not fetched.

`@RelatedTo(type=None, direction=Direction.OUTGOING, multiple=false, restrictType=false)` The field represents a relation to another or the same model.

`type` has to be set to the relationship type name.
`direction` defaults to an outgoing relationship, but can be also incoming or even both.
`multiple` allows multiple relationships to the same model.
`restrictType` restricts the type. Used when the same relationship type is used for multiple model relationships.

`@RelationshipCount(type=None, direction=Direction.OUTGOING, otherModel=Model.class)` The field should contain the relationship count for a specified relationship type when fetched. Will not be persisted.

`type` has to be set to the relationship type name.
`direction` defaults to an outgoing relationship, but can be also incoming or even both.
`otherModel` is used to only query for specific models.

`@Required` The field is required upon storing to database. If it is missing an exception will be thrown.

`@Unique` The field is unique across all models of the same type. Will be indexed as well. If a duplicate value is passed an exception will be thrown.

`@Uuid` Auto-generated uuid field. This should not be required as theres always an uuid field on each model.

2.3 Repositories

Repositories are used to communicate with the underlying database. We won't need to create a repository for each *model*. There's a default implementation, that will be used if no appropriate repository was found which provides basic functionality.

The naming of the repository is important, so that Frogr can find it. Names should start with the *model* name and end with "Repository" (case-sensitive) and it should extend `BaseModelRepository` or `BaseRelationshipRepository`.

If, for example, we want a repository for the `Person` model, we would create a repository called `PersonRepository`:

```java
public class PersonRepository extends BaseModelRepository<Person> {
  public void init() {
    Person rick = new Person("Rick Sanchez");
    Person beth = new Person("Beth Smith");
    Person jerry = new Person("Jerry Smith");
    Person morty = new Person("Morty Smith");
    Person summer = new Person("Summer Smith");
    // we need to save the people first, before we can create relationships
    save(rick, beth, jerry, morty, summer);

    rick.setChildren(Arrays.asList(beth));
    beth.setChildren(Arrays.asList(morty, summer));
    jerry.setChildren(Arrays.asList(morty, summer));
    save(rick, beth, jerry, morty, summer);
    MarriedWith marriedWith = new MarriedWith(jerry, beth);
    marriedWith.setYears(10L);
    service().repository(MarriedWith.class).save(marriedWith);
  }
}
```

We can access our repository easily by calling the `.repository(..)` method of the `Service` instance. The method takes either the *model* class or the name as string. There's access to it in any REST service class and inside any repository:

```java
@Path("person")
public class Persons extends CRUDService<PersonRepository, Person> {
  @GET
  @Path("init")
  public void init() {
    // insert some data
    try(Transaction tx = service().beginTx()) {
      if(repository().search().count() == 0) {
        repository().init();
        tx.success();
      }
    }
  }

  @GET
  @Path("custom-search")
  public List<Person> customSearch(@SearchParam SearchParameter params) {
    try(Transaction ignored = service().beginTx()) {
      return repository().search().params(params).list();
    }
  }
}
```

2.4 Services

Services define the REST entry points of our application. They should only have minimal logic and communicate with the repositories instead.

Services can extend the CRUDService class, which provides basic CRUD methods for creating, reading, updating and deleting models. They also can extend the RestService class, which only provides some minimal convenience methods. But we can also create a service from scratch.

Here's a basic example of a service, used for the Person models:

```
@Path("persons")
public class Persons extends CRUDService<PersonRepository, Person> {
  @GET
  @Path("init")
  public void init() {
    // insert some data
    try(Transaction tx = service().beginTx()) {
      if(repository().search().count() == 0) {
        repository().init();
        tx.success();
      }
    }
  }
}
```

And here's the repository implementation:

```
public class PersonRepository extends BaseModelRepository<Person> {
  public void init() {
    Person rick = new Person("Rick Sanchez");
    Person beth = new Person("Beth Smith");
    Person jerry = new Person("Jerry Smith");
    Person morty = new Person("Morty Smith");
    Person summer = new Person("Summer Smith");
    // we need to save the people first, before we can create relationships
    save(rick, beth, jerry, morty, summer);

    rick.setChildren(Arrays.asList(beth));
    beth.setChildren(Arrays.asList(morty, summer));
    beth.setMarriedWith(jerry);
    jerry.setChildren(Arrays.asList(morty, summer));
    jerry.setMarriedWith(beth);
    save(rick, beth, jerry, morty, summer);
  }
}
```

This would create these REST paths:

```
POST    /persons (de.whitefrog.frogr.example.rest.Persons)
GET     /persons (de.whitefrog.frogr.example.rest.Persons)
GET     /persons/{uuid: [a-zA-Z0-9]+} (de.whitefrog.frogr.example.rest.Persons)
POST    /persons/search (de.whitefrog.frogr.example.rest.Persons)
PUT     /persons (de.whitefrog.frogr.example.rest.Persons)
DELETE  /persons/{uuid: [a-zA-Z0-9]+} (de.whitefrog.frogr.example.rest.Persons)
GET     /persons/init (de.whitefrog.frogr.example.rest.Persons)
```

The `POST` and `PUT` methods both take a json object, representing the model to create or update.

The `GET` method takes url parameters used for a search operation. See *Searches* for further details. Additionally there's also a `GET /{uuid}` method for convenience, which searches for a particular person.

We can also use `POST /search` path, which takes the search parameters as a json object, but I strongly recommend using GET for that purpose, as only GET can be cached correctly.

`DELETE` has the UUID inside its path and will delete the model with that UUID.

We can also see our previously implemented `/init` path configured.

We use primarily UUIDs to reference models, not Ids as they can be reused by the underlying Neo4j database.

2.5 Searching

2.5.1 Java

In Java code there's a easy to use method in each repository. Here are some examples:

```
List<Person> results = search()
  .uuids(uuid1, uuid2)
  .fields(Person.Name, Person.MarriedWith)
  .list();

// Get a count of persons, where on of its parents name is "Jerry Smith".
long count = search()
  .filter(new Filter.Equals("parents.name", "Jerry Smith"))
  .count();

// Get a paged result of all persons, with a page size of 10, ordered by the name␣
↪property.
List<Person> page = search()
  .limit(10)
  .page(1)
  .orderBy(Person.Name)
  .fields(Person.Name)
  .list();

// Get a single person and its children with their names.
Person beth = search()
  .filter(Person.Name, "Beth Smith")
  .fields(FieldList.parseFields("name,children.name"))
  .single();
```

2.5.2 REST

Over HTTP you would normally use a `CRUDService`, that provides the neccessary methods, but we can of course write our own ones. Here are some examples, that would return the same results as the Java queries above:

```
// Filter results by uuids and return the name and the person married with the found␣
↪person.
http://localhost:8282/persons?uuids=e4633739050611e887032b418598e63f,
↪e4635e4a050611e88703efbc809ff2fd&fields=name,marriedWith
```

```
{
  "success": true,
  "data": [
    {
      "uuid": "e4633739050611e887032b418598e63f",
      "type": "Person",
      "name": "Beth Smith",
      "marriedWith": {
        "uuid": "e4635e4a050611e88703efbc809ff2fd",
        "type": "Person"
      }
    },
    {
      "uuid": "e4635e4a050611e88703efbc809ff2fd",
      "type": "Person",
      "name": "Jerry Smith",
      "marriedWith": {
        "uuid": "e4633739050611e887032b418598e63f",
        "type": "Person"
      }
    }
  ]
}
```

```
// Get a count of persons, where on of its parents name is "Jerry Smith".
http://localhost:8282/persons?filter=parents.name:=Jerry%20Smith&count
```

```
{
  "success": true,
  "total": 2,
  "data": [
    {
      "uuid": "e463d37c050611e887034f42b099b0cd",
      "type": "Person"
    },
    {
      "uuid": "e463ac6b050611e887038de1cbd926c1",
      "type": "Person"
    }
  ]
}
```

```
// Get a paged result of all persons, with a page size of 10, ordered by the name␣
↪property.
http://localhost:8282/persons?limit=10&page=1&order=name&fields=name
```

```
{
  "success": true,
  "data": [
    {
      "uuid": "e4633739050611e887032b418598e63f",
      "type": "Person",
      "name": "Beth Smith"
    },
    {
      "uuid": "e4635e4a050611e88703efbc809ff2fd",
```

(continues on next page)

(continued from previous page)

```
      "type": "Person",
      "name": "Jerry Smith"
    },
    {
      "uuid": "e463ac6b050611e887038de1cbd926c1",
      "type": "Person",
      "name": "Morty Smith"
    },
    {
      "uuid": "e4607818050611e8870361190053d169",
      "type": "Person",
      "name": "Rick Sanchez"
    },
    {
      "uuid": "e463d37c050611e887034f42b099b0cd",
      "type": "Person",
      "name": "Summer Smith"
    }
  ]
}
```

```
http://localhost:8282/persons?filter=name:=Beth%20Smith&fields=name,children.name
```

```
{
  "success": true,
  "data": [
    {
      "uuid": "e4633739050611e887032b418598e63f",
      "type": "Person",
      "name": "Beth Smith",
      "children": [
        {
          "uuid": "e463ac6b050611e887038de1cbd926c1",
          "type": "Person",
          "name": "Morty Smith"
        },
        {
          "uuid": "e463d37c050611e887034f42b099b0cd",
          "type": "Person",
          "name": "Summer Smith"
        }
      ]
    }
  ]
}
```

2.5.3 Usage in services

If we want to write a method that takes its own search parameters, we can use the `@SearchParam` annotation along with a `SearchParameter` argument:

```
@Path("person")
public class Persons extends CRUDService<PersonRepository, Person> {
  @GET
```

(continues on next page)

(continued from previous page)

```
  @Path("custom-search")
  public List<Person> customSearch(@SearchParam SearchParameter params) {
    try(Transaction ignored = service().beginTx()) {
      return repository().search().params(params).list();
    }
  }
}
```

2.5.4 Parameters

These are the possible querystring parameters, but you can find nearly identical methods in Java too.

uuids Comma-seperated list of uuids to search.

query Searches all indexed fields for a query string.

count Add a total value of found records, useful if the result is limited.

start Start returning results at a specific position, not required when `page` is set.

limit Limit the results.

page Page to return. Takes the limit parameter and sets the cursor to the needed position.

filter/filters Filter to apply. Filters start with the field name, followed by a **:** and the comparator. Valid comparators are:

= Equal
! Not equal
< less than
<= less or equal than
> greater than
>= greater or equal than
`({x}-{y})` in a range between x and y

fields Comma-seperated list of fields to fetch. Can also fetch sub-fields of related models seperated by a **.**, for example `children.name` would fetch all childrens and their names. Multiple sub-fields can be fetched inside curly braces, `children.{name,age}` would fetch all childrens and their names and ages.

return/returns Returns a related model instead of the service model.

order/orderBy/sort Comma-seperated list of fields on which the results are sorted. - before the field sorts in descending, + in ascending direction. If bypassed ascending direction is used.

2.6 Authorization and Security

2.6.1 Setup

Maven

```
<dependency>
    <groupId>de.whitefrog</groupId>
    <artifactId>frogr-auth</artifactId>
    <version>0.2.3-SNAPSHOT</version>
</dependency>
```

User Model

The User model has to extend `BaseUser` and defines our user, which can be passed in *Service* methods using the `@Auth` annotation.

```
class User : BaseUser() {
  @JsonView(Views.Secure::class)
  @RelatedTo(type = RelationshipTypes.FriendWith)
  var friends: ArrayList<User> = ArrayList()
}
```

As you can see the annotation `@JsonView(Views.Secure::class)` is used on the `friends` field. These views can be used on `Service` methods too, and describe what can be seen by the user. The default is `Views.Public::class`, so any field annotated with that `@JsonView` is visible to everyone. Fields without `@JsonView` annotation are always visible.

`BaseUser` provides some commonly used fields describing a user in an authentication environment:

You can write your own User class, but then you'll have to create your own oAuth implementation.

Warning: Be cautious with `Views.Secure` on `Service` methods, as it could reveal sensitive data. So it's best to have custom methods like `findFriendsOfFriends` for example to get all friends of the users friends instead of the common search function.

User Repository

Next, we'll have to define a repository for our users, extending `BaseUserRepository`:

```
public class UserRepository extends BaseUserRepository<User> {
  public String init() {
    String token;
    PersonRepository persons = service().repository(Person.class);
    if(search().count() == 0) {
      Person rick = new Person("Rick Sanchez");
      Person beth = new Person("Beth Smith");
      Person jerry = new Person("Jerry Smith");
      Person morty = new Person("Morty Smith");
      Person summer = new Person("Summer Smith");
      // we need to save the people first, before we can create relationships
      persons.save(rick, beth, jerry, morty, summer);

      rick.setChildren(Arrays.asList(beth));
```

(continues on next page)

(continued from previous page)

```
      beth.setChildren(Arrays.asList(morty, summer));
      beth.setMarriedWith(jerry);
      jerry.setChildren(Arrays.asList(morty, summer));
      jerry.setMarriedWith(beth);
      persons.save(rick, beth, jerry, morty, summer);

      User user = new User();
      user.setLogin("justin_roiland");
      user.setPassword("rickandmorty");
      user.setRole(Role.Admin);
      save(user);

      // login to create and print an access token - for test purposes only
      user = login("justin_roiland", "rickandmorty");
      token = "access_token=" + user.getAccessToken();
      System.out.println("User created. Authorization: Bearer " + user.
↪getAccessToken());
    } else {
      User user = login("justin_roiland", "rickandmorty");
      token = "access_token=" + user.getAccessToken();
      System.out.println("Authorization: Bearer " + user.getAccessToken());
    }
    return token;
  }
}
```

The extended class `BaseUserRepository` provides some basic functionality and security.

`register(user)` Registration of a new user, passwords will be encrypted by default.

`login(login, password)` Login method, encrypts the password automatically for you.

`validateModel(context)` Overridden to ensure a password and a role is set on new users.

Application

In our applications `run` method, we need to set up some authentication configurations:

```
public class MyApplication extends Application<Configuration> {
  private MyApplication() {
    // register the rest classes
    register("de.whitefrog.frogr.example.oauth");
    // register repositories and models
    serviceInjector().service().register("de.whitefrog.frogr.example.oauth");

  }

  @Override
  @SuppressWarnings("unchecked")
  public void run(Configuration configuration, Environment environment) throws␣
↪Exception {
    super.run(configuration, environment);

    Authorizer authorizer = new Authorizer(service().repository(User.class));
    AuthFilter oauthFilter = new OAuthCredentialAuthFilter.Builder<User>()
      .setAuthenticator(new Authenticator(service().repository(User.class)))
      .setAuthorizer(authorizer)
```

(continues on next page)

(continued from previous page)

```
      .setPrefix("Bearer")
      .buildAuthFilter();

    environment.jersey().register(RolesAllowedDynamicFeature.class);
    environment.jersey().register(new AuthValueFactoryProvider.Binder<>(User.class));
    environment.jersey().register(new AuthDynamicFeature(oauthFilter));
  }

  @Override
  public String getName() {
    return "frogr-auth-example-rest";
  }

  public static void main(String[] args) throws Exception {
    new MyApplication().run("server", "config/example.yml");
  }
}
```

Inside the `run` method, we set up `RolesAllowedDynamicFeature`, which activates the previously used `@RolesAllowed` annotation. We also set up `AuthValueFactoryProvider.Binder` which activates the later described `@Auth` injection annotation and `AuthDynamicFeature` which activates the actual oAuth authentication.

Services

Here's a simple service, that can only be called when the user is authenticated. The user will be passed as argument to the method:

```
@Path("person")
public class Persons extends AuthCRUDService<PersonRepository, Person, User> {
  @GET
  @Path("find-morty")
  @RolesAllowed(Role.User)
  public Person findMorty(@Auth User user) {
    try(Transaction ignored = service().beginTx()) {
      return repository().findMorty();
    }
  }
}
```

See how the `@RolesAllowed(Role.User)` annotation is used, to only allow this method to registered users. You can always extend the Role class and use your own roles. Predefined roles are `Admin`, `User` and `Public`.

The first (and only) parameter on `findMorty` is annotated with `@Auth` and has the type of our `User` class created before. This will inject the currently authenticated user and also tells the application that this method is only allowed for authenticated users.

The extended class `AuthCRUDService` provides some convenient methods for authentication and sets some default `@RolesAllowed` annotations on the basic CRUD methods. All predefined methods are only allowed for registered and authenticated users.

`authorize` Override to implement your access rules for specific models. Is used by default in create and update methods.

`authorizeDelete` Override to implement your rules to who can delete specific models. Is used by default in delete method.

CHAPTER 3

License

CHAPTER 4

Contact

Questions? Please contact jochen@whitefrog.de

CHAPTER 5

Help

If you have any questions, contact jochen@whitefrog.de