Glagol DSL Documentation

Release 0.5

Yoan-Alexander Grigorov

May 22, 2018

Getting started

CHAPTER 1

Introduction

1.1 What is Glagol DSL?

Glagol DSL is a domain specific language with the goal to help developers engineer microservices using the Domain-Driven Design[1] concepts out-of-the-box. To achieve this, Glagol DSL incorporates object-oriented structures that are much closer to the concepts of Model-Driven Design[2] than the usual object-oriented languages. Additionally, Glagol also supports basic web (api) framework capabilities like routing, request handling and controllers.

Glagol DSL includes grammatical declarations for Entities, Value Objects, Repositories and Controllers. In general, all of those are embedded into the language's syntax and their usage implies following rules based on the concept behind each of them.

1.2 How does Glagol DSL work?

Glagol DSL is a transpiler and the output language is PHP. Simply put, the Glagol DSL sources (*.g) are parsed, type-checked and then compiled into PHP source code. To achieve most of its functional capabilities the Glagol DSL environment relies strongly on the Lumen framework from Laravel and the Doctrine 2 ORM.

Architecturally, Glagol DSL is a simple client-server application. The server app is the actually compiling your sources and the client is used to send compile requests and more. This approach provides performance benefits and it also allows the compiler to run over a network (Compiler-as-a-service).

1.3 Under the hood

Glagol DSL is mainly written on Rascal MPL for the server app and Java for the client app.

[1] Domain-driven design: Tackling complexity in the heart of software, Eric Evans, 2004, Addison-Wesley Professional.
[2] Model-Driven Design is a set of design patterns and part of Domain-Driven Design as described by Eric Evans[1]

CHAPTER 2

Installation

Before we create a simple Glagol DSL microservice lets setup Glagol DSL!

2.1 Prerequisites

2.1.1 Java

Make sure you have java version 1.8 installed before you can use Glagol DSL:

```
$ java -version
java version "1.8.0_151"
Java(TM) SE Runtime Environment (build 1.8.0_151-b12)
Java HotSpot(TM) 64-Bit Server VM (build 25.151-b12, mixed mode)
```

2.2 Get the server app

Download the `glagol-dsl-server-0.6.3.jar` from the latest release:

```
$ wget https://github.com/glagol-dsl/glagol-dsl/releases/download/0.6.3/glagol-dsl-
↪server-0.6.3.jar ~/glagol-dsl-server.jar
```

Then you can alias the `glagol-server` as a command so that it can be used easily. To do that you simply add to the end of `~/.bashrc` (or `~/.zshrc`) the following command:

```
alias glagol-server='java -Xmx400m -jar /full/path/to/glagol-dsl-server.jar'
```

And then reload your rc:

```
$ source ~/.bashrc
```

or for zsh:

```
$ source ~/.zshrc
```

2.3 Get the client app

Similarly to the server app the client app jars are available on Github releases. Download the latest release jar:

```
$ wget https://github.com/glagol-dsl/glagol-dsl-client/releases/download/0.3.7/glagol-
↪dsl-client-0.3.7.jar ~/glagol-dsl-client.jar
```

Then you can alias the `glagol` as a command so that it can be used easily. To do that you simply add to the end of `~/.bashrc` (or `~/.zshrc`) the following command:

```
alias glagol='java -jar /full/path/to/glagol-dsl-client.jar'
```

Attention: When using the client as of version >= 0.4, for now you have to run compile with the *-l* (–no-ssh-auth) flag until the ssh feature is completed.

And then reload your rc:

```
$ source ~/.bashrc
```

or for zsh:

```
$ source ~/.zshrc
```

2.4 Start the Glagol DSL server

Glagol DSL server app can be initiated using the following command:

```
$ glagol-server daemon
```

The expected output is the following:

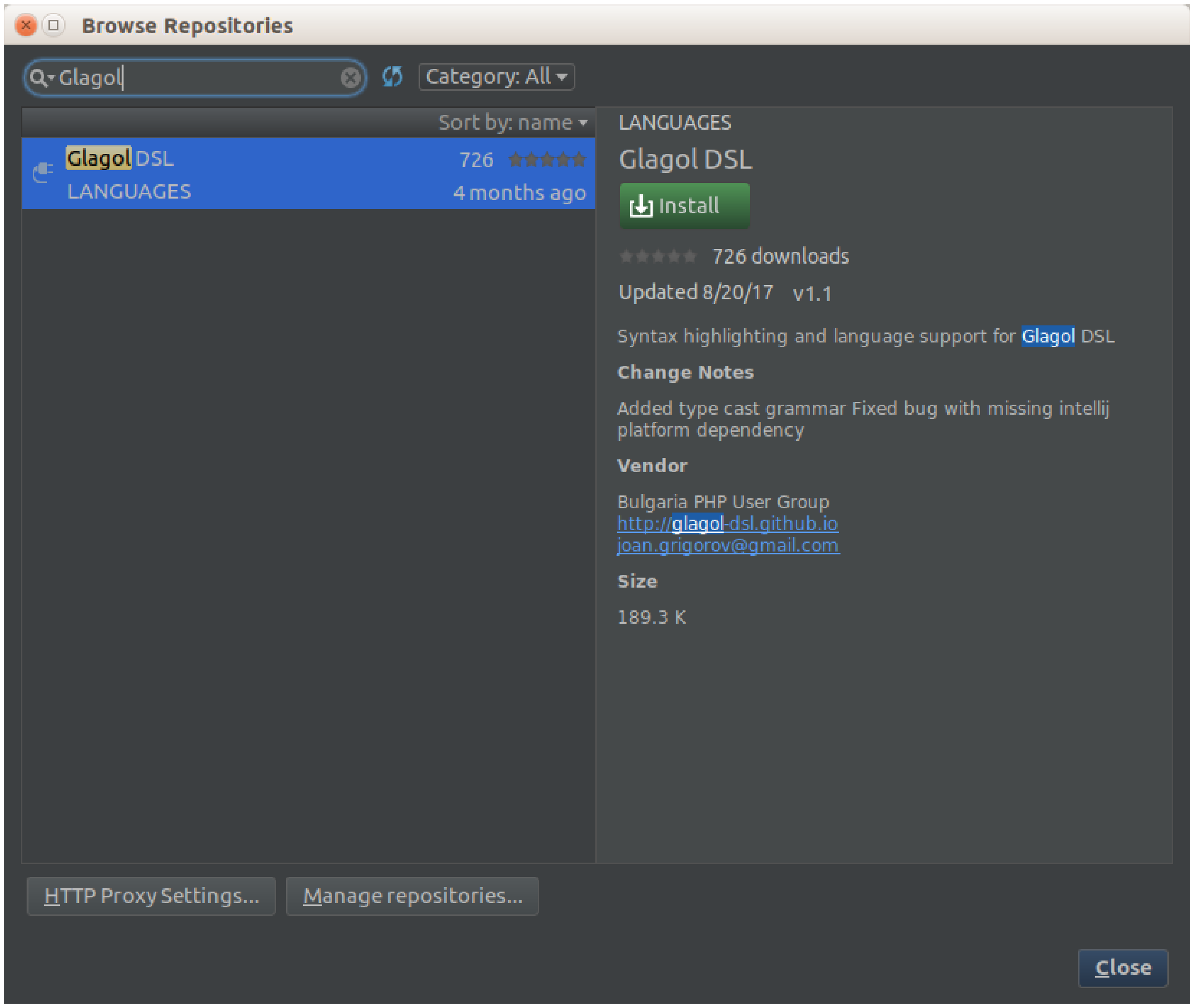
```
Initializing Glagol...
Daemon listens on port 51151...
```

The initialization phase might take up to 10 seconds depending on your machine's performance. You need to keep the Glagol DSL server running if you wish to compile.

Next section will explain more about setting up an initial project structure, compiling and running an app.

2.5 Get the IDE plugin

Glagol DSL supports an IntelliJ IDEA (PhpStorm, WebStorm, etc.) plugin. You can find in in the JetBrains repositories
.



726 downloads

Updated 8/20/17 v1.1

Syntax highlighting and language support for Glagol DSL

Change Notes

Added type cast grammar Fixed bug with missing intellij platform dependency

Vendor

Bulgaria PHP User Group
http://glagol-dsl.github.io
joan.grigorov@gmail.com

Size

189.3 K

CHAPTER 3

Setup new project (with Docker)

After a successful *Installation* of Glagol DSL server and client apps you are ready to start developing. As discussed within the installation section the fastest way to start developing is to use Docker and Docker Compose.

3.1 Create a new project structure

Simply clone the default project skeleton:

```
$ mkdir glagol-hello-world
$ git clone git@github.com:glagol-dsl/docker-compose-project-skeleton.git glagol-
↪hello-world
```

Attention: You need to have the Glagol DSL Server app running before you proceed. Remember, you can start the server app by simply running `$ glagol-server daemon`.

After you clone the default project skeleton you can compile it for first time! Simply run the compile command:

```
$ glagol compile
```

The output should be something like:

```
Successfully compiled glagol project
```

And then you are ready to spin-up the Docker Compose environment:

```
$ docker-compose up
```

Hint: The Docker Compose configuration will automatically install all PHP Composer dependencies upon containers startup. In fact, it will do so every time you spin-up the containers. See `./scripts/composer.sh` for how it works under the hood.

The credentials for connecting to the mysql database can be found in `docker-compose.yml`. By default they are:

Host	localhost
port	3307
database	glagol
user	root
password	123

Next you can proceed directly to *Hello world controller*.

CHAPTER 4

Setup new project (without Docker)

This section describes how you can manually setup your environment for developing with Glagol DSL.

Warning: Skip this chapter if you already have setup your project using Docker.

4.1 Install PHP, MySQL and a Web Server

You need PHP 7.1 and MySQL 5.7 at the very least. The choice of a web server is not crucial, both nginx and apache were tested and proven to work perfectly fine.

4.1.1 PHP DS extension

Glagol DSL data structures rely on the ds extension. The simplest way to install is using pecl:

```
$ sudo pecl install ds
$ sudo phpenmod ds
```

4.1.2 PHP Pdo extensions

Currently, Glagol DSL supports only the MySQL database as a possible ORM backend. Therefore, the `pdo_mysql` extension is required. On Ubuntu/Debian you can install the MySQL PDO driver by typing:

```
$ sudo apt-get install php-mysql
```

Alternatively, you can also install it using pecl:

```
$ sudo pecl install pdo pdo_mysql
$ sudo phpenmod pdo pdo_mysql
```

4.1.3 PHP mbstring extension

Lumen framework requires the mbstring PHP extension. On Ubuntu you can install it simply by typing:

```
$ sudo apt-get install php-mbstring
```

Or alternatively using pecl:

```
$ sudo pecl install mbstring
$ sudo phpenmod mbstring
```

4.1.4 Composer

Finally, you need to make sure that you have composer installed. On Ubuntu you can install it using the package manager:

```
$ sudo apt-get install composer
```

Hint: If you want to significantly speed-up your composer you might consider installing the global performance boost package hirak/prestissimo: `composer global require hirak/prestissimo`

4.2 Create a new project structure

Lets create a new project. For the purpose we need a new directory:

```
$ mkdir glagol-hello-world
```

Navigate inside it and create two sub-directories `src` and `out`:

```
$ cd glagol-hello-world
$ mkdir src
$ mkdir out
```

The `src` folder will host the Glagol `*.g` source files, and the `out` directory will be used for the compiled sources.

Attention: You need to have the Glagol DSL Server app running before you proceed. Remember, you can start the server app by simply running `$ glagol-server daemon`.

Next, create a new package by making a new directory within the `src` folder:

```
$ mkdir src/HelloWorld
```

At the end your directory structure should look like this:

```
.
├── out
└── src
    └── HelloWorld
```

Important: Make sure that your Web Server's virtual host maps to `out/public` folder. This is where the `index.php` is landing and where the routing starts.

Next, you need to compile for first time! Simply run the compile command:

```
$ glagol compile
```

The output should be something like:

```
Successfully compiled glagol project
```

4.3 Install Composer dependencies

The final step is to install the PHP Composer packages:

```
$ composer install --prefer-dist -d out/
```

Hint: PHP compiled sources are generated in the `./out/` directory of your project's root. Make sure to map your web server's root to `./out/public` directory.

4.4 Set environment variables

The final step is to create an `./out/.env` file that will hold environment variables (such as app's name and MySQL connection credentials). Here is an example contents:

```
APP_NAME="Glagol DSL Sandbox"
APP_ENV=local
APP_DEBUG=true
DB_HOST=localhost
DB_PASSWORD=123
DB_USERNAME=root
DB_DATABASE=glagol
```

CHAPTER 5

Hello world controller

After you have your directory structure set up it is time to create your first endpoint - the hello world page. For the purpose create a file `src/HelloWorld/HelloWorldController.g` with the following contents:

```
namespace HelloWorld

rest controller /hello-world {
    index {
        return "Hello world!";
    }
}
```

The snippet above represents an HTTP controller that is set to handle the index page of the `/hello-world` route by responding with a `Hello world!` message.

The `index` declaration is in fact the action method that is handling the GET request.

Furthermore, you might have noticed that the controller declaration does not have a name but only a route. Although controllers in Glagol DSL do look like classes they are a special type of structure. For example, controllers cannot be instantiated by the developer and they carry the routing information with them. More about controllers can be found in the language reference guide.

5.1 Compile the sources

Now compile the project! Simply run the compile command from your project's root directory:

```
$ glagol compile
```

The output should be something like:

```
Successfully compiled glagol project
```

Hint: PHP compiled sources are generated in the `./out/` directory of your project's root. If you are not using the preconfigured Docker Compose project skeleton you need to map your web server's root to `./out/public` directory.

5.2 Testing the index

The Docker Compose default project skeleton will run nginx on port `8081` (if you installed a fresh nginx manually the port is probably `80`):

```
$ curl localhost:8081/hello-world
```

The microservice should respond with:

```
$ curl localhost:8081/hello-world
Hello world!
```

CHAPTER 6

First Entity

One the most fundamental patterns of enterprise software nowadays is the Entity. In general, an entity is simply an object that is used to persist data and state during runtime. Moreover, entities are usually distinguished by some form of identity. For example, in the case with a relational database table an entity can be identified by a primary key field and its value.

Glagol DSL supports a built-in syntax declaration for entities that can be automatically bundled with database tables. Generally speaking, the purpose of having a built-in language structure for entities is to hide the underlying application logic from the developer and bring the focus on the entity as a business item itself.

Usually, entities are maintained with the help of the so called *Object-relational mapper* libraries. Doctrine 2 ORM is one of the popular ORMs in the PHP world. Therefore, it is being used by the Glagol DSL compiled codes. However, a Glagol DSL developer does not need to worry about ORMs because this part is taken care by the language and the compiler itself.

6.1 Create your first entity

Lets create a Glagol DSL entity. Create a new package by making a directory named `src/MusicLib`. Next, put the following code inside:

```
namespace MusicLib

entity Song {
    int id;
    string title;
    string genre;
    string author;
}
```

Save this file as `src/MusicLib/Song.g`. Notice that the file name and path needs to match the defined namespace and entity name. Otherwise, the compiler will notify with an error that there is naming inconsistency.

To show a use cases of this entity lets create a new fresh controller for the purpose `src/MusicLib/SongController.g`:

```
namespace MusicLib

rest controller /song {
    index {
        return new Song();
    }
}
```

Compile the sources using `$ glagol compile` and then `$ curl localhost:8081/song`. The response should display the following json:

```
{
    "id": null,
    "title": null,
    "genre": null,
    "author": null
}
```

6.2 Constructors

Entities can have constructors. They are defined in a Java-like way:

```
namespace MusicLib

entity Song {
    int id;
    string title;
    string genre;
    string author;

    Song(string songTitle) {
        title = songTitle;
    }
}
```

Constructors must use the name of the entity. Additionally, you can override constructors as long as they do not have duplicating signatures (the combination of arguments being passed):

```
namespace MusicLib

entity Song {
    int id;
    string title;
    string genre;
    string author;

    Song(string songTitle) {
        title = songTitle;
    }

    Song(string song, string author) {
        title = song;
        this.author = author;
    }
}
```

Notice how the second constructor uses `this` to assign property values. This is because the argument names match the property names. Unlike PHP, properties can be accessed directly from the scope of any method (including constructors). Only when an argument has the same name as a defined property you need to use `this.propertyName` notation to differentiate the property from the parameter within the same scope.

Try to compile the app. What you will get is a type-check error like this:

```
Cannot compile, errors found:
[/src/MusicLib/SongController.g:5] Cannot match constructor Song()
```

This is because you do not have a constructor that accepts no argument as used in the controller. Lets add some parameters:

```
namespace MusicLib

rest controller /song {
    index {
        return new Song("Virus", "Marko Markovic Brass Band");
    }
}
```

This instantiation will use the second constructor because of the matching signature (`string, string`). Note that glagol only accepts double quotes `"` for strings and not single quotes.

Next, compile the app and after you `$ curl localhost:8081/song` the output will be:

```
{
    "id": null,
    "title": "Virus",
    "genre": null,
    "author": "Marko Markovic Brass Band"
}
```

6.3 Guards

Functional languages like Haskell support the concept of function *guards*. A guard is simply a boolean expression that is triggered before the function's body. Furthermore, if the guard expression evaluates as `true` then the function logic will be executed. Otherwise, the next override declaration with the same signature will be checked until success. If no overriding qualifies for execution an error is thrown.

Glagol DSL implements guards that can be applied to constructors and methods. Lets modify our Song entity in a way it uses guards:

```
namespace MusicLib

entity Song {
    int id;
    string title;
    string genre;
    string author;

    Song(string title, string author) {
        this.title = title;
        this.author = author;
        this.genre = "Balkan";
    } when author == "Marko Markovic Brass Band";
```

(continues on next page)

(continued from previous page)

```
    Song(string title, string author) {
        this.title = title;
        this.author = author;
        this.genre = "Jazz";
    } when author == "Miles Davis";
}
```

Obviously, if you instantiate the entity as `new Song("Virus", "Marko Markovic Brass Band")` the genre will be set to 'Balkan'. The second constructor has a guard that will evaluate as true if the artist is 'Miles Davis', and as a result it will set the genre to 'Jazz'.

In general, you can think of guards as a way to structurally avoid if-else statements. Their only purpose in Glagol DSL is to reduce code volume and enhance readability. Otherwise, guards work is any other typical conditional structure.

You might have noticed something in the example above, and that is that there is no *fallback* constructor yet - a one with the same signature but with no guard. In a case where we instantiate a `Song` entity and the author is neither `Marko Markovic Brass Band` or `Miles Davis` what would be the outcome? Logically - it should fail with an error, and this is exactly the way Glagol DSL will behave too. Lets update the controller with the following code:

```
namespace MusicLib

rest controller /song {
    index {
        return new Song("Soul Power 74", "Maceo Parker");
    }
}
```

Since there is not constructor to handle the entity instantiation the app will respond with an error message like this:

```
{
    "message": "Sorry, something went wrong",
    "error": "Cannot match constructor for Song",
    "trace": [
        {
            "file": "src/MusicLib/SongController.g",
            "line": 5,
            "name": null
        }
    ]
}
```

Glagol DSL will try to match methods/constructors with guards first. Furthermore, you can see that errors that originally happening in the generated PHP are mapped to the actual `*.g` source files.

Lets default the song genre by adding a constructor without a guard:

```
namespace MusicLib

entity Song {
    int id;
    string title;
    string genre;
    string author;

    Song(string title, string author) {
        this.title = title;
```

(continues on next page)

(continued from previous page)

```
        this.author = author;
        this.genre = "Balkan";
    } when author == "Marko Markovic Brass Band";

    Song(string title, string author) {
        this.title = title;
        this.author = author;
        this.genre = "Jazz";
    } when author == "Miles Davis";

    Song(string title, string author) {
        this.title = title;
        this.author = author;
        this.genre = "Unknown";
    }
}
```

The next chapter explains how to read, persist and delete entities from the database.

6.4 Relationships

Glagol DSL also supports database relationships. Say we have a `genres` table that the `songs` table has a foreign key relation to from a field `genre_id`.

```
mysql> SELECT * FROM glagol.songs;
+----+-------+----------+---------------------------+
| id | title | genre_id | author                    |
+----+-------+----------+---------------------------+
|  1 | Virus |        1 | Marko Markovic Brass Band |
+----+-------+----------+---------------------------+

    mysql> SELECT * FROM glagol.genres;
+----+--------+
    | id | name   |
+----+--------+
    |  1 | Balkan |
+----+--------+
```

If you create a Genre entity, you can type hint it in the Song entity and Glagol DSL will automatically lazy load it.

```
namespace MusicLib

@table="genres"
entity Genre {
        @id
        @sequence
        int id;

        string name;
}
```

```
namespace MusicLib

entity Song {
```

(continues on next page)

(continued from previous page)

```
    int id;
    string title;
    Genre genre;
    string author;
    }
```

Querying songs will now result in

```
{
    "id": null,
    "title": "Virus",
    "genre": {
        "id": 1,
        "name": "Balkan"
    },
    "author": "Marko Markovic Brass Band"
}
```

CHAPTER 7

Repositories and entities

In the previous chapter we investigated some basics of creating and using entities. However, the examples so far were leaving the entities static - without persistence with a database.

7.1 Prepare the database

Lets create a database table for our `MusicLib::Song` entity:

```
USE `glagol`;

CREATE TABLE `songs` (
  `id` INT NOT NULL AUTO_INCREMENT,
  `title` VARCHAR(45) NULL,
  `genre` VARCHAR(45) NULL,
  `author` VARCHAR(45) NULL,
  PRIMARY KEY (`id`));
```

Attention: If you are using the Docker Compose default project structure you can just put the SQL snippet displayed above in the `./database/data.sql` file. The database folder is imported by the MySQL docker container upon build - just rebuild your stack and the table will be created and ready to use. Additionally, you can tweak the database name the app is using by changing the `DB_DATABASE` environmental variable from `docker-compose.yml`. The database name for any Glagol DSL app defaults to `glagol`.

7.2 Annotating the entity

Next step is to tell Glagol DSL that the `Song` entity related to the `glagol.songs` database table.

First, add the `@table` annotation just before the declaration of the entity:

```
namespace MusicLib

@table="songs"
entity Song {
    int id;
    string title;
    string genre;
    string author;
}
```

After you compile the sources the environment will know that the `Song` entity is bundled with the `glagol.songs` table.

Secondly, we need to annotate the primary key field and flag it as auto-incremented. Simply add the `@id` and `@sequence` annotations just before the `int id;` property declaration:

```
namespace MusicLib

@table="songs"
entity Song {
    @id
    @sequence
    int id;
    string title;
    string genre;
    string author;
}
```

The `@id` annotation will tell Glagol DSL that the targeted field is the primary key field. As for the `@sequence` annotation - it will indicate that the field is set for auto-increment.

Compile the sources using the already familiar `glagol compile` command.

Now we told Glagol DSL about all the metadata required to bundle the entity with an existing database table.

7.3 Songs repository

Lets insert some data into the database using the Song entity!

First, we need a Repository object. In general, repositories are a special type of objects that hold the sole responsibility of supporting the lifecycle of entities. Moreover, a Repository is a frequently used enterprise design pattern which has been repeatedly publicised by a number of influential authors such as Martin Fowler and Eric Evans throughout the last two decades. To put it simply, a repository can persist(create, update), remove and retrieve entities from the data storage.

Glagol DSL provides a built-in syntax declaration for repositories:

```
namespace MusicLib

repository for Song {

}
```

Save this code as `src/MusicLib/SongRepository.g`.

Lets investigate the code snippet. From `repository for Song` we can conclude that repositories are, in a way, *attached* to entities. Additionally, Glagol DSL allows only one repository per entity (one-on-one relationship) to be created - doing otherwise will result in typecheck errors during compilation. Last but not least, the source file name has to follow the mandatory naming convention of `<Entity>Repository.g` where `<Entity>` is the name of the targeted entity.

So far so good. However, our new repository does not do anything yet. Lets change that by introducing a `save` method:

```
namespace MusicLib

repository for Song {
    public void save(Song song) {
        persist song;
        flush;
    }
}
```

Yep - it is *that* simple. Repositories expose the `persist` and `flush` statements. The first one tells Glagol DSL that we want to initiate the persistence of the object that is being passed. Secondly, `flush` statement will save all the data into the data storage. Think of it this way - `persist` will tell Glagol DSL *"Hey, here is an object for you to keep an eye on!"*, and `flush` says *"Bring my changes to the database, please!"*.

Note: Entity persisting and flushing is inspired and based on Doctrine 2 ORM's way of operating with entities.

Finally, lets save our Song entity into the database! Modify the `SongController.g` to use the repository:

```
namespace MusicLib

rest controller /song {

    repository<Song> songs = get repository<Song>;

    index {
        Song balkanSong = new Song("Virus", "Marko Markovic Brass Band");

        songs.save(balkanSong);

        return balkanSong;
    }
}
```

Lets investigate this piece of code. The very first thing to notice is `repository<Song> songs = get repository<Song>;` - in this line we simply define a new property `songs` which is of type Song repository type. Moreover, you lets focus on two main keypoints here:

- `repository<Song>` - this is the syntax used to address repository types;
- `get repository<Song>` - this indicates that the property should be automatically set with the repository instance for a value.

Hint: The line `repository<Song> songs = get repository<Song>;` can be simplified to `repository<Song> songs = get selfie;`. In general, the **selfie** keyword will reflect the type declared in the property (in this case `repository<Song>`).

Now just compile once again and test using `curl localhost:8081/song`! You will see a response just similar to:

```
{
    "id": 1,
    "title": "Virus",
    "genre": "Balkan",
    "author": "Marko Markovic Brass Band"
}
```

Notice that this time there is a value assigned to the `id` field. Generally speaking, this is the first indicator that the data was successfully inserted into the data storage. In fact, lets query MySQL directly to see the stored data:

```
mysql> SELECT * FROM glagol.songs;
+----+-------+--------+---------------------------+
| id | title | genre  | author                    |
+----+-------+--------+---------------------------+
|  1 | Virus | Balkan | Marko Markovic Brass Band |
+----+-------+--------+---------------------------+
1 row in set (0.00 sec)
```

7.4 Removing entities

Very similarly to persisting entities we can also remove such. However, instead of the `persist` statement Glagol DSL provides the `remove` counterpart:

```
namespace MusicLib

repository for Song {
    public void save(Song song) {
        persist song;
        flush;
    }

    public void remove(Song song) {
        remove song;
        flush;
    }
}
```

Notice, that flushing the requested changes is still necessary if you want to delete the entity records.

In the next chapter we are going to look into how to query entities using Glagol DSL's embedded query language!

CHAPTER 8

Querying

So far we learnt how to create and persist entities but what about querying for such? Long story short, Glagol DSL uses a built-in SQL dialect for the purpose. In general, there are a few things to consider about the Glagol Query Language:

- Queries can only be utilized within repository methods
- Queries target **entities** not database tables
- Queries are executed immediately
- Queries are a type of expressions

8.1 Finding all entities

We start by creating a `findAll` method in the Song repository:

```
namespace MusicLib

repository for Song {
    public void save(Song song) {
        persist song;
        flush;
    }

    public void remove(Song song) {
        remove song;
        flush;
    }

    public Song[] findAll() {
        Song[] songs = SELECT s[] FROM Song s;

        return songs;
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Secondly, lets look closely at what we did here. Obviously, the `SELECT s FROM Song s` is a Glagol Query. One can easily recognize the great similarities between SQL and Glagol QL. However, think of it this way - Glagol Query Language is used to ask for entities where SQL usually queries database tables directly. Therefore, the result of a Glagol Query is either an entity or a collection of entities.

Lets break down the query from above. The first part is `SELECT s[]`. Simply put, we say that we want to select a list of `s`'s where `s` is just an alias defined in the `FROM` part. Furthermore, `FROM Song s` is exactly the part where we first select the target entity and then we alias it so it can be referenced in other parts of the query.

Notice how the selection part says `s[]`. In case you remove the brackets then the query will return the first entity directly. Hence, when you want to select the whole result set then you simply add the brackets.

Thirdly, since queries are just expressions we can refactor the bit from above and simply return the query expression directly:

```
namespace MusicLib

repository for Song {
    public void save(Song song) {
        persist song;
        flush;
    }

    public void remove(Song song) {
        remove song;
        flush;
    }

    public Song[] findAll() {
        return SELECT s[] FROM Song s;
    }
}
```

Finally, in order to see how the query works we simply use it in the controller:

```
namespace MusicLib

rest controller /song {

    repository<Song> songs = get selfie;

    index {
        return songs.findAll();
    }
}
```

Last thing to do is to compile using `$ glagol compile`. Now simply `curl localhost:8081/song` and you will see a list of song objects!

8.2 Querying for a single entity

Next task is to create a typical `find(int id)` method that will query for just one entity. You guessed it - we need a `WHERE` clause:

```
namespace MusicLib

repository for Song {
    public void save(Song song) {
        persist song;
        flush;
    }

    public void remove(Song song) {
        remove song;
        flush;
    }

    public Song[] findAll() {
        return SELECT s[] FROM Song s;
    }

    public Song find(int id) {
        return SELECT s FROM Song s WHERE s.id = <<id>>;
    }
}
```

As mentioned earlier, if we want to select only one entity we simply write `SELECT s`. Secondly, lets investigate the `WHERE s.id = <<id>>` clause. It is obvious that we access a field from the entity by targeting the `s` alias. The right side of the equation indicates that we embed a Glagol DSL expression. In the example above we simply pass the `id` parameter that is provided by the method. However, we can use any expression we like there (like `WHERE s.id = <<id + 232>>`). The << and >> are just delimiters that indicate the start and end of an embedded expression.

In order to show how the finder method works lets modify our sandbox controller in the following way:

```
namespace MusicLib

rest controller /song {

    repository<Song> songs = get selfie;

    index {
        return songs.find(1);
    }
}
```

Compile the source and simply `curl localhost:8081/song` to test as usual.

In the next chapter we are going to learn how to put everything together under a unified Rest API using the controller.

CHAPTER 9

Rest controllers

The examples so far always relied on modifications of the index page of the Song controller. Although it is easy for displaying tutorial examples this approach is highly inappropriate for real-world scenarios. Luckily, Glagol DSL provides a built-in Rest API support in controllers.

9.1 Index action

Every rest controller supports the standard Rest actions which are based on the well-known HTTP 1.1 request methods. As you might have noticed already, so far we used the `index` action for testing. Generally speaking, when we want to run the index we simply make a `GET` request to the route that the controller defines. The response can be any value. Usually, the Rest index will return a list of resources. In our case, this fits with the result of querying for all entities. Therefore, lets modify our Songs index to select all song entities:

```
namespace MusicLib

rest controller /song {

    repository<Song> songs = get selfie;

    index {
        return songs.findAll();
    }
}
```

We already had this exact bit in the previous tutorial. Lets keep it this way!

9.2 Show action

What if we want to request a single resource? For this task Glagol DSL provides the developers with a `show` action. Lets introduce it:

```
namespace MusicLib

rest controller /song {

    repository<Song> songs = get selfie;

    index {
        return songs.findAll();
    }

    show (int id) {
        return songs.find(id);
    }
}
```

The main difference is that the show action accepts an argument which is used as a criteria for finding a resource. In the example above we rely on the existing `find` repository method that will do the job.

Now compile the sources and call the following curl command:

```
$ curl localhost:8081/song/1
```

Naturally, the `show` action is initiated by a `GET` request method in combination with an argument appended to the end of the route. Therefore, a `show` request URI has the `/song/{parameter}` format.

However, there is an alternative way of finding entities when we rely on the primary key field that is annotated with `@id`. For the purpose, we can even avoid our manually created `find` method or the repository altogether. To put it simply, developers can utilize the `@autofind` annotation to the first parameter of the action like this:

```
namespace MusicLib

rest controller /song {

    repository<Song> songs = get selfie;

    index {
        return songs.findAll();
    }

    show (@autofind Song song) {
        return song;
    }
}
```

Lets look at the differences. Instead of accepting `int id` as a parameter, we type `Song song` preceded by the `@autofind` annotation. This very annotation tells Glagol DSL to attempt to find the entity by primary key field and the value of the parameter being passed with the request.

Important: The annotation approach does not require you to have a `find` repository method whatsoever. It is a built-in behavior provided the Glagol DSL runtime environment.

9.3 Store action

The `store` method is there to handle `POST` requests aimed to create new resources. Additionally, the `Glagol::Http::Request` object is necessary for extracting input `POST` data from the request:

```
namespace MusicLib

import Glagol::Http::Request;

rest controller /song {

    repository<Song> songs = get selfie;

    index {
        return songs.findAll();
    }

    show (@autofind Song song) {
        return song;
    }

    store (Request request) {
        Song song = new Song(request.input("title"), request.input("author", "unknown
↪"));

        songs.save(song);
        return song;
    }
}
```

Do not forget to compile the source!

Hint: The `Glagol::Http::Request` object is provided by the Glagol DSL standard library. It provides the following methods for extracting `POST` data:

```
string input(string key);
string input(string key, string default);
```

Use the `string default` parameter to provide a default value if the `string key` does not exist in the request payload.

Since Glagol DSL relies on Lumen Framework for its runtime both json and x-www-form-urlencoded payloads are supported. For this example we are going to use a json payload:

```
$ curl -X POST -d '{"title":"Born to raise hell", "author": "Motorhead"}' -H "Content-
↪Type: application/json" localhost:8081/song
```

9.4 Update action

The `update` action is similar to both show and store actions in a way. First, just like `show` it requires a parameter by which to identify a resource. Secondly, just like `store` it can use the `input()` methods from `Glagol::Http::Request`.

In contrast to both actions, the HTTP request method for `update` is `PUT`. Lets look at this example:

```
namespace MusicLib

rest controller /song {

    repository<Song> songs = get selfie;

    index {
        return songs.findAll();
    }

    show (@autofind Song song) {
        return song;
    }

    store (Request request) {
        Song song = new Song(request.input("title"), request.input("author", "unknown
↪"));

        songs.save(song);
        return song;
    }

    update (Request request, @autofind Song song) {
        song.setTitle(request.input("title"));
        song.setAuthor(request.input("author", "unknown"));

        songs.save(song);

        return song;
    }
}
```

Additionally, the example above requires two setter methods in the entity:

```
namespace MusicLib

@table="songs"
entity Song {
    // ... properties and constructors from before...

    public void setTitle(string title) {
        this.title = title;
    }

    public void setAuthor(string author) {
        this.author = author;
    }
}
```

Compile the sources and test with curl:

```
$ curl -X PUT -d '{"title":"Killed by death", "author": "Motorhead"}' -H "Content-
↪Type: application/json" localhost:8081/song/2
```

9.5 Delete action

Glagol DSL provides the `delete` action to handle `DELETE` HTTP requests:

```
namespace MusicLib

rest controller /song {

    repository<Song> songs = get selfie;

    index {
        return songs.findAll();
    }

    show (@autofind Song song) {
        return song;
    }

    store (Request request) {
        Song song = new Song(request.input("title"), request.input("author", "unknown
↪"));

        songs.save(song);
        return song;
    }

    update (Request request, @autofind Song song) {
        song.setTitle(request.input("title"));
        song.setAuthor(request.input("author", "unknown"));

        songs.save(song);

        return song;
    }

    delete (@autofind Song song) {
        songs.remove(song);
    }
}
```

Compile the sources and test with curl:

```
$ curl -X DELETE localhost:8081/song/1
```

9.6 Create and edit actions

Additionally, you can also implement the `create` and `edit` actions that are basically responding to `GET` requests but with additional route extensions.

First, the `create` action is usually used to return an initial state entity. Typically this functionality is used to retrieve a blank resource from the service:

```
namespace MusicLib

rest controller /song {
```

(continues on next page)

(continued from previous page)

```
    repository<Song> songs = get selfie;

    index {
        return songs.findAll();
    }

    show (@autofind Song song) {
        return song;
    }

    store (Request request) {
        Song song = new Song(request.input("title"), request.input("author", "unknown
↪"));

        songs.save(song);
        return song;
    }

    update (Request request, @autofind Song song) {
        song.setTitle(request.input("title"));
        song.setAuthor(request.input("author", "unknown"));

        songs.save(song);

        return song;
    }

    delete (@autofind Song song) {
        songs.remove(song);
    }

    create {
        return new Song("Please, enter song title", "Please, enter song author (band)
↪");
    }

    edit (@autofind Song song) {
        return song;
    }
}
```

To get the response from this endpoint you need to append `/create` to the route like this:

```
$ curl localhost:8081/song/create
```

Similarly, `edit` is just like `show` but it requires the `/edit` extension to be accessed:

```
$ curl localhost:8081/song/1/edit
```

CHAPTER 10

Indices and tables

- genindex
- search