WB Games Programming Exercise

Steven Furlong, 2017/03/27

Interpretation and Discussion of Requirements

There are two ways of interpreting requirements documents: that they are solid gold and should be implemented exactly as written, or that they are an approximation of what needs to be built and are likely to have inconsistencies and other errors.

I am taking the second approach here, largely because of “create, update, and modify user records”. Updating and modifying would be the same thing in 4-feature CRUD, suggesting some inaccuracy in the requirements. “Update” and “modify” are sometimes distinguished, with both modifying existing records if one exists and one creating a new record if it doesn't exist. However, there is some disagreement whether update or modify will create records. Also, reading existing records was not mentioned in the requirements.

If possible I'd ask the BA or client for clarification. If that wasn't possible, I'd check with my boss on which way to interpret the requirement. If that wasn't possible, I'd interpret the requirements loosely and implement what seemed to be the most useful APIs for my best understanding of the use cases.

All that said, it's possible that "create, update, and modify user records" is exactly what was wanted. It's conceivable that, for instance, a record archive system would need to create and update records but not allow them to be deleted or read. Even in this interpretation I'd still need to talk to the BA or client to find out the difference between updating and modifying records.

As stated, I'm applying a loose interpretation to the requirements. The prototype I built provides full 4-feature CRUD capability, as well as getting an index listing of all records in the system. This was in large part a result of my implementing the prototype in Groovy/Grails and getting full functionality essentially for free.

Data Model

Generally the database can be set up as tightly defined or with a looser definition. Tightly defined databases will typically run on an RDBMS and have lookup tables, foreign key constraints, and triggers to maintain consistency and to enforce business logic.

Loosely defined databases can also run on RDBMSs but can also run in MongoDB and similar systems. Consistency is generally less of a concern, and enforcing business rules is up to the non-DBMS program code. A simple, loose schema is often useful in the requirements exploration and prototyping stages of development because it's quicker and easier to make something workable and usually easier to change. This is also useful in production when writes are infrequent compared to reads and the overhead of the consistency mechanism is relatively expensive.

Most of my experience is with tightly constrained database schemas, and I'm inclined to design systems this way. Having the checks defined in the database schema avoids many inconsistency nightmares.

See the attached ERD. It shows three “real” tables and two relationship tables.

Implementation

The prototype project is done in Groovy/Grails, version 3.2.8. I chose this framework because it's excellent for quickly making a prototype web application or web service backed by a database.

I defined three domain classes, Users, Games, and Achievements, with Users being defined as having lists of Games and Achievements. I defined all three classes with the

```
@Resource(uri='/users')
```

annotation (adjusting the uri to fit)

Grails then set up Hibernate mappings for all tables as well as for relationships between Users and the other tables. This in effect create M:N join tables as shown in the ERD, as well as foreign key constraints which prevent Games and Achievements records from being deleted if they are mentioned in any User record.

Note that even though the generated schema has join tables and foreign key constraints, in effect it is a “loose” model, as described above. There are no master lists of possible games or achievements which limit what can be entered into a user's data. Instead, the caller of the API can put in whatever is needed in creating records.

The Achievements table has only a single defined field, achievement. As such, the table was not really needed and the Users table could have had a list of Strings. I defined the table because it seemed likely that fields would be added in the future, such as the date the achievement was attained.

Because of the @Resource annotation, Grails set up RESTful APIs for each of the domain classes. Each class has the following interfaces:

• index (list all)	GET with no ID
• read	GET with the record ID
• create	POST with Postdata
• update	PUT with a record ID and Postdata
• delete	DELETE with a record ID

These features are all provided automatically, with no coding needed other than defining the domain classes' fields and putting in @Resource. They can be suppressed entirely (eg, disable the DELETE command), conditionally allowed, or renamed.

I also set up a few Users, Games, and Achievements records, to have something to work with in development and debugging.

API

These web services are RESTful.

The default return type is JSON. All API calls can return XML by appending “.xml”.

/users/1

```
{"id":1,"achievements":[],"active":false,"email":"john@foo.org","games":[],"name":"John Doe"}
```

/users/1.xml

```
<?xml version="1.0" encoding="UTF-8"?><users id="1"><achievements /><active>false</active><email>john@foo.org</email><games /><name>John Doe</name></users>
```

View all users

/users

Get a list of all users in the database.

Output:

```
[{"id":1,"achievements":[],"active":false,"email":"john@foo.org","games":[],"name":"John Doe"},
{"id":3,"achievements":[{"id":1}],"active":true,"email":"player3@thegame.com","games":
[{"id":1}],"name":"Player 3"}]
```

Notes:

Use this interface with caution, as it will return *all* users. This is intended for development, with small databases, and should not be exposed to general callers.

View one user

View the information for a single user in the database.

/users/1

Successful output:

```
{"id":1,"achievements":[],"active":false,"email":"john@foo.org","games":[],"name":"John Doe"}
```

Error output:

```
{"message":"Not Found","error":404}
```

The ID was not found.

Create a user

Create a user record in the database.

POST /users

postdata {"name":"Player 3","email":"player3@thegame.com","achievements":[{"id":1},{"id":4},
{"id":5}],"games":[],"active":true}

Successful output:

```
{"id":4,"name":"Player 3","email":"player3@thegame.com","achievements":[{"id":1},{"id":4},
{"id":5}],"games":[],"active":true}
```

Error output:

{"message":"Internal server error","error":500}

> This normally means that a consistency constraint was violated, which usually means specifying a game or achievement ID which doesn't exist.
>
> Examine the stack trace in the server's output to confirm this.

Update a user

Change elements of a user's data.

PUT /users/1

postdata {"name":"Player 1 is gone","email":"none","achievements":[],"games":[],"active":false}

Successful output:

{"id":1,"name":"Player 1 is gone","email":"none","achievements":[],"games":[],"active":false}

Error output:

{"message":"Not Found","error":404}

> The identified user does not exist

{"message":"Internal server error","error":500}

> This normally means that a consistency constraint was violated, which usually means specifying a game or achievement ID which doesn't exist.
>
> Examine the stack trace in the server's output to confirm this.

Delete a user

Remove a user from the database.

DELETE /users/1

Successful output:

None

Error output:

{"message":"Not Found","error":404}

> Specified user ID is not found.

Notes:

Use this with caution. The record is deleted, not simply marked as inactive.

View all games

/games

Get a list of all games in the database.

Output:

[{"id":1,"level":"champ","losses":0,"name":"Monopoly","score":100.0,"wins":10}]

Notes:

Use this interface with caution, as it will return *all* games. This is intended for development, with small databases, and should not be exposed to general callers.

View one game

View the information for a single game in the database.

/games/1

Successful output:

```
{"id":1,"level":"champ","losses":0,"name":"Monopoly","score":100.0,"wins":10}
```

Error output:

```
{"message":"Not Found","error":404}
```

The ID was not found.

Create a game

Create a game record in the database.

POST /games

postdata {"name":"Shoot-em-up 1","score":100.0,"level":"n00b","wins":3,"losses":6}

Successful output:

```
{"id":21,"name":"Shoot-em-up 1","score":100.0,"level":"n00b","wins":3,"losses":6}
```

Error output:

```
{"message":"Internal server error","error":500}
```

Examine the stack trace in the server's output to find the problem.

Update a game

Change elements of a game's data.

PUT /games/1

postdata {"name":"strategy 1","score":0,"level":"none","wins":0,"losses":0}

Successful output:

```
{"id":1,"name":"strategy 1","score":0,"level":"none","wins":0,"losses":0}
```

Error output:

```
{"message":"Not Found","error":404}
```

The identified game does not exist

```
{"message":"Internal server error","error":500}
```

Examine the stack trace in the server's output to determine the problem.

Delete a game

Remove a game from the database.

DELETE /games/1

Successful output:

None

Error output:

{"message":"Not Found","error":404}

> Specified game ID is not found.

{"message":"Internal Server Exception","error":500}

> This usually means that an integrity constraint was violated, usually because the record was referred to by a User record.

Notes:

Use this with caution. The record is deleted, not simply marked as inactive.

View all achievements

/achievements

Get a list of all achievements in the database.

Output:

[{"id":1,"achievement":"Skunk Da Competition!"},{"id":2,"achievement":"Rule Them All!"},{"id":3,"achievement":"Pwn Da Luzers!"}]

Notes:

Use this interface with caution, as it will return *all* achievements. This is intended for development, with small databases, and should not be exposed to general callers.

View one achievement

View the information for a single achievement in the database.

/achievements/1

Successful output:

{"id":1,"achievement":"Skunk Da Competition!"}

Error output:

{"message":"Not Found","error":404}

> The ID was not found.

Create a achievement

Create an achievement record in the database.

POST /achievements

postdata {"achievement":"5 wins in a row"}

Successful output:

{"id":21,"achievement":"5 wins in a row"}

Error output:

{"message":"Internal server error","error":500}

> Examine the stack trace in the server's output to find the problem.

Update a achievement

Change elements of an achievement's data.

PUT /achievements/1

postdata {"achievement":"5 wins in a row"}

Successful output:

{"id":1,"achievement":"5 wins in a row"}

Error output:

{"message":"Not Found","error":404}

> The identified achievement does not exist

{"message":"Internal server error","error":500}

> Examine the stack trace in the server's output to determine the problem.

Delete a achievement

Remove an achievement from the database.

DELETE /achievements/1

Successful output:

None

Error output:

{"message":"Not Found","error":404}

> Specified user ID is not found.

{"message":"Internal Server Exception","error":500}

> This usually means that an integrity constraint was violated, usually because the record was referred to by a User record.

Notes:

Use this with caution. The record is deleted, not simply marked as inactive.

Suggested Additional Interfaces and Enhancements

Find user by name

Find user by email

Show all users with a specified achievement

Show all users with activity on a specified game

Show game and achievement values rather than simply IDs when listing users.

Allow for creating users with game and achievement values in place, rather than requiring callers to create Game or Achievement records, get the IDs, and then fill in the IDs when creating or updating a User record.

> In both cases, the coding was a bit of a challenge. Treating this project as the first prototype for showing users, I viewed the current functionality as useful for getting initial feedback while being very quick and cheap to put together.

Change the URI for retrieving or updating a user's game or achievement data to something like

```
/users/1/games/4
```

Tests

I manually tested with curl on the command line, such as

```
curl -i -X PUT -H "Content-Type: application/json" -d '{"achievements":[{"id":1},{"id":4}]}'
localhost:8081/users/1
```

This could be automated very simply with a shell script, sending known input and scraping the output, or any number of test frameworks could do the job. I did not implement any of that because, as above, I am viewing this as an initial prototype service with many changes expected.

I did not write any unit tests because so far there is no business logic to be tested.

Technology Stack and Runtime Environment

This prototype needs Grails 3.2.8 and Java 8. There should be no other dependencies and it should run on any operating system which has both of those. When run in “dev” mode, some sample data will be preloaded into a dummy database, enough for smoke testing and demoing the prototype to users.

Beyond this implementation, a number of technology stacks could easily support simple web services. For example:

- LAMP
- Java
- Groovy/Grails

Which to choose depends on the expected size and complexity of the code, the complexity of the final schema, the number of records in the tables, performance requirements, and team expertise.