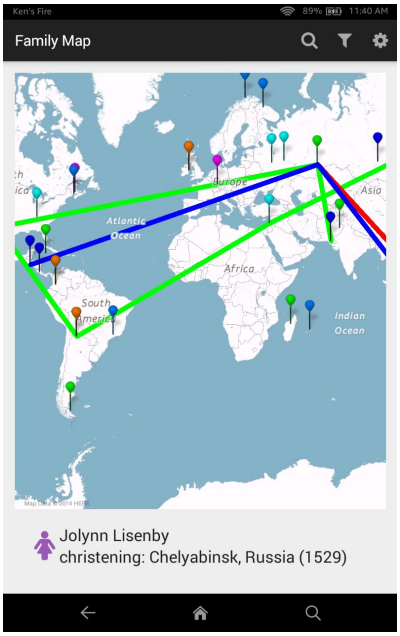
Family Map Server Specification

Acknowledgements

The Family Map project was created by Jordan Wild. Thanks to Jordan for this significant contribution.

Family Map Introduction

Family Map is an application that provides a geographical view of your family history. One of the most exciting aspects of researching family history is discovering your origins. Family Map provides a detailed view of where you came from. It does so by displaying information about important events in your ancestors' lives (birth, marriage, death, etc.), and plotting their locations on a Google or Amazon map.



Family Map uses a client/server architecture, which means it consists of two separate programs (a "client" program and a "server" program). The Family Map client is an Android app that lets a user view and interact with their family history information (see image on the left). The Family Map server is a regular, non-Android Java program that runs at a publicly-accessible location in the "cloud" (although, for development purposes you can run it locally on your laptop or some other machine). When a user runs the Family Map client app, they are first asked to log in. After authenticating the user's identity with the Family Map server, the client app retrieves the user's family history data from the server. The server is responsible for maintaining user accounts as well as dispensing family history data for Family Map users. Family Map's client/server architecture is typical of many real-world applications that you are probably familiar with (Facebook, Twitter, etc.)

In CS 240 you will design and implement both parts of the Family Map application. In Project 1 you will create your Family Map server. In Project 2 you will create your Family Map Android client. (NOTE: In the real world you would typically develop the client and server simultaneously rather than sequentially, but for pedagogical reasons we will ask you to create them sequentially.)

Family Map Server (FMS)

This project focuses on designing, implementing, and testing the Family Map server. The server is a regular Java program. When it runs the server does not display an interactive user interface (i.e., it is "headless"). Rather than interacting with a user, it waits for Family Map clients to connect to it over the Web for the purposes of authenticating user logins and retrieving user family history information. Although it has no user interface, the server may display diagnostic or informational messages in the console and/or in a log file. This can be useful for debugging the server and monitoring its operation. Additionally, because we do not have real family history data for our users, part of the server's functionality is to generate artificial family history data for each Family Map user.

The purpose of this project is to learn about and gain experience with the following:

- Designing, implementing, and testing a large, multi-faceted program
- Relational database concepts and programming
- Creation of server programs that publish web APIs
- Automated unit and integration testing

Data Definitions

Users

In order to use Family Map, one must first create a user account. Your server should store information about each user account in its database. The following properties should be associated with each user account:

Username: Unique user name (non-empty string)
Password: User's password (non-empty string)
Email: User's email address (non-empty string)
First Name: User's first name (non-empty string)
Last Name: User's last name (non-empty string)
Gender: User's gender (string: "f" or "m")
Person ID: Unique Person ID assigned to this user's generated Person object - see Family History Information section for details (non-empty string)

Authorization Tokens

When a user logs in to your server, the login request sent from the client to the server must include the user's username and password. If login succeeds, your server should generate a unique "authorization token" string for the user, and return it to the client. Subsequent requests sent from the client to your server should include the auth token so your server can determine

which user is making the request. This allows non-login requests to be made without having to include the user's credentials, thus reducing the likelihood that a hacker will intercept them. For this scheme to work, your server should store auth tokens in its database and record which user each token belongs to. Also, to protect against the possibility that a hacker might intercept a user's auth token, it is important that each new login request generate and return a unique auth token. It should also be possible for the same user to be logged in from multiple clients at the same time, which means that the same user could have multiple active auth tokens simultaneously.
An auth token should be included in the HTTP "Authorization" request header for all requests that require an auth token.

Command-Line Arguments

Your server should accept the following command-line arguments:

1. Port number on which the server will accept client connections. This value is an integer in the range 1-65535. EX: 8080

Family History Information

Your server should generate and store in its database family history information for each user. This data should include two types of objects: Persons and Events. A user's family history data should include at least one Person object, which represents the user him or herself. Beyond the user's own Person object, there can be zero or more additional generations of data. When your server is asked to generate family history data for a user, it will be told how many generations of ancestor data to create (0 generations = the user, 1 generation = the user + parents, 2 generations = the user + parents + grandparents, etc.). Based on the requested number of generations, you should fill out the user's family tree with generated Person and Event data.

For each Person you should generate a set of Event objects that describe important events from the person's life. The types of events that can be generated is open-ended, but would typically include events for birth, baptism, marriage, and death, but could also include events for christening, additional marriages, etc. The generated events should be made as realistic as possible. For example, a person should not die before they are born, they should not get married when they are three years old, death should be the last event in their life, etc. Event locations may be randomly selected, or you can try to make them more realistic (e.g., many people live their lives in a relatively small geographical area).

The course web site provides some files you may use to help generate person, country, and city names and locations.

Persons

Each generated Person should have the following properties:

Person ID: Unique identifier for this person (non-empty string)
Descendant: User (Username) to which this person belongs
First Name: Person's first name (non-empty string)
Last Name: Person's last name (non-empty string)
Gender: Person's gender (string: "f" or "m")
Father: ID of person's father (possibly null)
Mother: ID of person's mother (possibly null)
Spouse: ID of person's spouse (possibly null)

Events

Each generated Event should have the following properties:

Event ID: Unique identifier for this event (non-empty string)
Descendant: User (Username) to which this person belongs
Person: ID of person to which this event belongs
Latitude: Latitude of event's location
Longitude: Longitude of event's location
Country: Country in which event occurred
City: City in which event occurred
EventType: Type of event (birth, baptism, christening, marriage, death, etc.)
Year: Year in which event occurred

Persistence

All of the data previously described should be stored in your server's database so it is not lost in case of a server reboot.

Web APIs

The primary function of the server is to publish a set of web APIs for use by the Family Map client. Your server should implement each of the following web API operations.

/user/register

URL Path: /user/register
Description: Creates a new user account, generates 4 generations of ancestor data for the new user, logs the user in, and returns an auth token.
HTTP Method: POST
Auth Token Required: No
Request Body:

```
{
        "userName": "susan",            // Non-empty string
```

```
	"password": "mysecret",		// Non-empty string
	"email": "susan@gmail.com", // Non-empty string
	"firstName": "Susan",		// Non-empty string
	"lastName": "Ellis",		// Non-empty string
	 "gender": "f"			// “f” or “m”
}
```
Errors: Request property missing or has invalid value, Username already taken by another user, Internal server error

Success Response Body:

```
{
	"authToken": "cf7a368f",	// Non-empty auth token string
	"userName": "susan",		// User name passed in with request
	"personID": "39f9fe46"			// Non-empty string containing the Person ID of the
					//	user’s generated Person object
}
```
Error Response Body:

```
{
	“message”: “Description of the error”
}
```

/user/login

URL Path: /user/login

Description: Logs in the user and returns an auth token.

HTTP Method: POST

Auth Token Required: No

Request Body:

```
{
	"userName": "susan",		// Non-empty string
	"password": "mysecret"		// Non-empty string
}
```
Errors: Request property missing or has invalid value, Internal server error

Success Response Body:

```
{
	"authToken": "cf7a368f",	// Non-empty auth token string
	"userName": "susan",		// User name passed in with request
	"personID": "39f9fe46"			// Non-empty string containing the Person ID of the
					//	user’s generated Person object
}
```
Error Response Body:

```
{
	“message”: “Description of the error”
}
```

/clear

URL Path: /clear
Description: Deletes ALL data from the database, including user accounts, auth tokens, and generated person and event data.
HTTP Method: POST
Auth Token Required: No
Request Body: None
Errors: Internal server error
Success Response Body:
```
{
	“message”: “Clear succeeded.”
}
```
Error Response Body:
```
{
	“message”: “Description of the error”
}
```

/fill/[username]/{generations}

URL Path: /fill/[username]/{generations}
Example: /fill/susan/3
Description: Populates the server's database with generated data for the specified user name. The required "username" parameter must be a user already registered with the server. If there is any data in the database already associated with the given user name, it is deleted. The optional “generations” parameter lets the caller specify the number of generations of ancestors to be generated, and must be a non-negative integer (the default is 4, which results in 31 new persons each with associated events).
HTTP Method: POST
Auth Token Required: No
Request Body: None
Errors: Invalid username or generations parameter, Internal server error
Success Response Body:
```
{
	“message”: “Successfully added X persons and Y events to the database.”
}
```
Error Response Body:
```
{
	“message”: “Description of the error”
}
```

/load

URL Path: /load
Description: Clears all data from the database (just like the /clear API), and then loads the posted user, person, and event data into the database.
HTTP Method: POST
Auth Token Required: No
Request Body: The “users” property in the request body contains an array of users to be created. The “persons” and “events” properties contain family history information for these users. The objects contained in the “persons” and “events” arrays should be added to the server’s database. The objects in the “users” array have the same format as those passed to the /user/register API with the addition of the personID. The objects in the “persons” array have the same format as those returned by the /person/[personID] API. The objects in the “events” array have the same format as those returned by the /event/[eventID] API.

```
{
	“users”: [ /* Array of User objects */ ],
	“persons”: [ /* Array of Person objects */ ],
	“events”: [ /* Array of Event objects */ ]
}
```

Errors: Invalid request data (missing values, invalid values, etc.), Internal server error
Success Response Body:

```
{
	“message”: “Successfully added X users, Y persons, and Z events to the database.”
}
```

Error Response Body:

```
{
	“message”: “Description of the error”
}
```

/person/[personID]

URL Path: /person/[personID]
Example: /person/7255e93e
Description: Returns the single Person object with the specified ID.
HTTP Method: GET
Auth Token Required: Yes
Request Body: None
Errors: Invalid auth token, Invalid personID parameter, Requested person does not belong to this user, Internal server error
Success Response Body:

```
{
	"descendant": "susan",		// Name of user account this person belongs to
```

```
        "personID": "7255e93e",     // Person's unique ID
        "firstName": "Stuart",      // Person's first name
        "lastName": "Klocke",       // Person's last name
        "gender": "m",              // Person's gender ("m" or "f")
        "father": "7255e93e"        // ID of person's father [OPTIONAL, can be missing]
        "mother": "f42126c8"        // ID of person's mother [OPTIONAL, can be missing]
        "spouse":"f42126c8"         // ID of person's spouse [OPTIONAL, can be missing]
}
```

Error Response Body:

```
{
        "message": "Description of the error"
}
```

/person

URL Path: /person
Description: Returns ALL family members of the current user. The current user is determined from the provided auth token.
HTTP Method: GET
Auth Token Required: Yes
Request Body: None
Errors: Invalid auth token, Internal server error
Success Response Body: The response body returns a JSON object with a "data" attribute that contains an array of Person objects. Each Person object has the same format as described in previous section on the /person/[personID] API.

```
{
        "data": [ /* Array of Person objects */ ]
}
```

Error Response Body:

```
{
        "message": "Description of the error"
}
```

/event/[eventID]

URL Path: /event/[eventID]
Example: /event/251837d7
Description: Returns the single Event object with the specified ID.
HTTP Method: GET
Auth Token Required: Yes
Request Body: None
Errors: Invalid auth token, Invalid eventID parameter, Requested event does not belong to this user, Internal server error

Success Response Body:

```
{
        "descendant": "susan"       // Name of user account this event belongs to (non-empty
                                    //    string)
        "eventID": "251837d7",      // Event's unique ID (non-empty string)
        "personID": "7255e93e",     // ID of the person this event belongs to (non-empty string)
        "latitude": 65.6833,        // Latitude of the event's location (number)
        "longitude": -17.9,         // Longitude of the event's location (number)
        "country": "Iceland",       // Name of country where event occurred (non-empty
                                    //    string)
        "city": "Akureyri",         // Name of city where event occurred (non-empty string)
        "eventType": "birth",       // Type of event ("birth", "baptism", etc.) (non-empty string)
        "year": 1912,               // Year the event occurred (integer)
}
```

Error Response Body:

```
{
        "message": "Description of the error"
}
```

/event

URL Path: /event

Description: Returns ALL events for ALL family members of the current user. The current user is determined from the provided auth token.

HTTP Method: GET

Auth Token Required: Yes

Request Body: None

Errors: Invalid auth token, Internal server error

Success Response Body: The response body returns a JSON object with a "data" attribute that contains an array of Event objects. Each Event object has the same format as described in previous section on the /event/[eventID] API.

```
{
        "data": [ /* Array of Event objects */ ]
}
```

Error Response Body:

```
{
        "message": "Description of the error"
}
```

Web API Test Page

To allow interactive testing of your server, we have created a Web API Test Page that lets you:

1. Interactively construct a Web API call
2. Send the Web API call to the server
3. See the output returned by the server for the call

This will be a useful development and testing tool for you, and will also be used by the TAs to pass-off your server.

The “home page” for your server should be the Web API Test Page. This means that when a user points a web browser at your server (e.g., http://localhost:8080/ OR http://localhost:8080/index.html), your server should return the Web API Test Page, thus allowing the user to interactively test your server. Your server’s Web API Test Page should be fully-functional (including images and styling).

All of the HTML, CSS, and image files that implement the Web API Test Page are provided on the course web site. These files should be placed in a folder within your server project. When your server receives an HTTP GET request asking for one of these files (based on the request URL), it should return the contents of the requested file in the body of the HTTP response. Doing this will allow a user to “browse” your server’s home page.

The end result will be that your server will perform two functions at once:

1. Service Web API requests from clients
2. Act like a normal web server by serving up its home page (the Web API Test Page)

In web applications, it is typical for a server to do both of these things.

Automated Tests

Use the JUnit testing framework to implement automated unit tests for your Data Access and Service classes.

You are expected to have at least 2 test cases for each public method found in these classes, typically one positive (or passing) and one negative (or failing). Some methods might not have the possibility of failing; in this case you should have 2 positive test cases. Review the following Q&A for more details.

Q: How many test cases do I need?
A: You need at least 2 test cases per public method found in your Service and DAO classes.

Q: Can I do more?
A: Absolutely. You may find doing more test cases to be helpful as a debugging tool.

Q: How much do I need to do for each test case?

A: Each test case should run through a complete execution of the method that it is testing and then use assertions to ensure that the method ran as expected. Most (if not all) test cases will involve multiple assertions.

Q: What does a negative (or failing) test case look like? Is it okay if my code crashes?
A: Your code should not crash. If your method is designed to throw a certain exception (perhaps a “UserAlreadyExistsException”) as part of its error reporting, that is okay. Just make sure to catch it in your test case. Some examples of negative test cases include trying to log in with a bad password, trying to find a personID that does not exist, or passing invalid parameters. This is just the beginning though. We want you to think about the possible weak points in your code, and try to find them. Make them as unique as possible; don’t just pass null into the parameters for every negative case. You can still lose points for bad test cases.

Q: Is it okay that my test cases depend on methods besides the one it tests?
A: Yes. We wish to avoid this where possible because it makes debugging easier when the test only relies on one method, but it’s fine to have to call a getter in order to check that your add method worked or to call the add method in order to have something to get. Just remember that you’re *stress-testing* the method that the test case is for, not the methods it uses to ensure correct operation.

Design

Two keys to succeeding on a large project like this are: 1) get started early, and 2) make a plan that breaks down the project into smaller pieces, and shows how those pieces will eventually fit together to make a working program. To help you do this effectively, you are required to submit a preliminary design for your project. Your design should include two parts:

1. Database Schema: Make a text file that contains all of the SQL CREATE TABLE statements needed to create all of the tables in your database schema.
2. Class Documentation: Create stub classes for your Model classes, Data Access classes, Service classes, and Request and Result classes. Document them with Javadoc comments. Run Javadoc to generate HTML documentation.

Your design should be submitted electronically. To do so, create a web site containing: 1) the text file with your SQL CREATE TABLE statements, and 2) a directory containing the files generated by Javadoc. Send an email to the TAs containing: 1) your name, 2) the URL of your SQL text file, and 3) the URL of the index.html file for your Javadoc documentation. The TAs will click on these URLs to view your design document. It is your responsibility to ensure that this works properly. If it does not, the TAs will be unable to grade your work.

(You might not be aware of this, but any files that you place in the public_html directory of your CS home directory are automatically published on the web. This allows you to easily create your

own personal website. For example, if a user has the CS login name fred, the URL for his web site is http://students.cs.byu.edu/~fred/. If he were to place a file named database.sql in his public_html directory, it would be accessible on the web at the URL http://students.cs.byu.edu/~fred/database.sql. You may use your CS website to submit your design document, or some other web site that you have.)

Source Code/Test Case Evaluation

After you pass off your project with a TA, you should immediately submit your project source code for grading. Your grade on the project will be determined by the date you submitted your source code, not the date that you passed off. If we never receive your source code, you will not receive credit for the assignment. Here are the instructions for submitting your project source code:

1. In Android Studio, execute “Build -> Clean Project” (if this option is not available, you may skip this step).
2. Create a ZIP file containing ALL of your project’s files (not just the Java files)
3. The name of the ZIP file should be your NetID. For example, if your NetID is “bob123”, the name of your zip file should be “bob123.zip”.
4. Submit your ZIP file through Learning Suite

To demonstrate that your test cases execute successfully, you should run your unit tests inside Android Studio, and make a screenshot of the successful test results displayed by Android Studio. Submit your screenshot through Learning Suite. This screenshot is submitted separately from your code ZIP file (i.e., they are different assignments in Learning Suite).

The following criteria will be used to evaluate your source code:

- ❏ (25%) Effective class, method, and variable names
- ❏ (30%) Effective decomposition of classes and methods
- ❏ (30%) Code layout is readable and consistent
- ❏ (15%) Effective organization of classes into Java packages

Grading

Your grade on this project will consist of the following components:

1. Design (quality of your preliminary design)
2. Functionality (how well your server works)
3. Test Case Quality (quality of your test cases, do they compile and run?, etc.)
4. Source Code Quality (quality of your source code)

See the course policies page on the CS 240 website for details on how much each of these components counts toward your overall course grade.

Additionally, students will have up to 5 passoff attempts for the Server before points will be deducted. This is an effort to encourage students to test their code before attempting to passoff. Click here for a general guideline of what the TA's will be testing in their passoff; use it as a starting point. You will want to test these suggestions extensively. You can assume that we will always give valid formatting to your web client. After your 5th failed passoff attempt you will lose 1% per attempt from your FMS Program grade.