FACULTY OF ENGINEERING AND ARCHITECTURE

DEPARTMENT INTEC
RESEARCH GROUP IBCN

Geavanceerde softwareontwikkeling

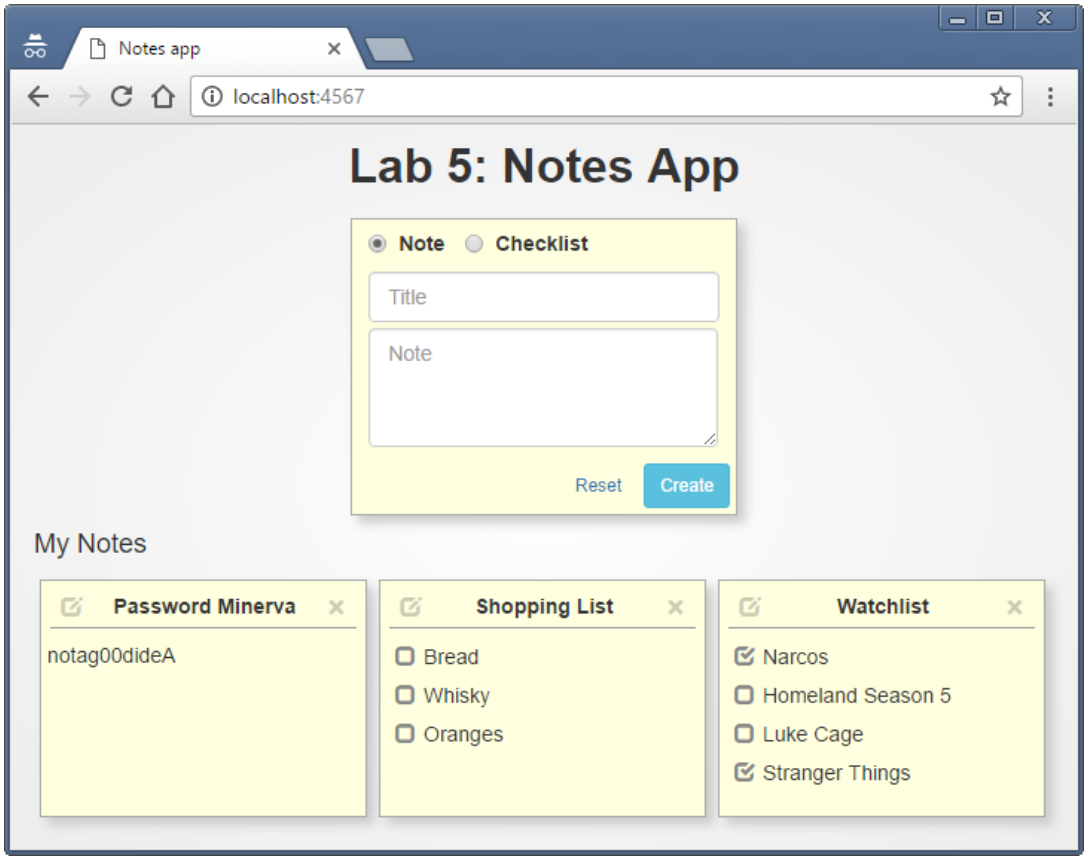
Lab Session V

Composite, Visitor, Proxy, Chain of Responsibility

GHENT UNIVERSITY

Introduction

The goal of this final Lab Session is to create a Web-based notebook application. Users will be able to authenticate with the backend server and then create, edit and delete note messages and checklists (which are persisted on the backend server file system) using the provided Web UI (see screenshot).



For each UI action, an HTTP request is sent from your browser to the backend Java server that you will be implementing. The handling of this request is based around the *Chain of Responsibility* pattern. Each incoming request is transferred along the handler chain until a handler is found that matches the request properties (such as HTTP method, path, etc…) and thus can process the request. Some handlers can halt the execution of the chain, e.g. when the user authentication that is sent along with the request is incorrect.

When a handler is found for the HTTP request, it will interact with our storage component to create, update or delete an entry on disk for the relevant note. The storage component will be used as a regular Map collection (containing key-value pairs for a unique String id that maps to a Note object). The storage implementation will then rely on the *Proxy* pattern to transparently synchronize the Map with a location on disk (where the server is running).

Each HTTP request will thus result in a Note object that is sent to the server or one or more Note objects that are received from the server. Unfortunately, we cannot directly send Java objects over the network (using HTTP). However, if we translate our Java objects to JSON (using the *Composite* structure from the JSON library you already familiarized yourselves with in Lab Session 1, we can transmit a text-representation to the client. To convert the Java objects to JSON we will use the *Visitor* pattern in combination with the provided JSON library. (The reverse conversion from JSON to Java objects is already implemented for you).

Dependencies

We've explained Maven in Lab Session 4 and how it facilitates using libraries in Java. For this Lab Session we won't start from scratch, the following list explains the various libraries that will be used:

- **IBCN JSON-lib** is our own utility library for interacting with JSON (Lab Session 1 was based on this). It allows you to read, write and manipulate JSON in Java.
- **Spark** is a micro framework for creating web applications in Java >8 with minimal effort. It enables us to handle HTTP request using the *Chain of Responsibility* pattern.
- **SLF4J** is a logging library that allows us to report application info or errors in a generic way (instead of printing statements to the console directly).
- **Apache Commons IO** is a library that provides a number of utility operations on top of the standard Java IO API.

Implementation

As for Lab Session 4, we provide you with a step-by-step guide of how the Notebook Web application can be implemented:

Step 1: Design the Visitor pattern

The model that is used to represent the notes is already implemented: the class `Note` represents a Note object with its unique id, title and content attribute. The content of a note is either a simple message containing a String (`Message.java`) or a checklist (`CheckList.java`) containing items that have a label and a checked/unchecked state (`CheckListItem.java`). There is also a Credentials class that is used to represent authentication info (username/password).

The goal for this step is to make the Note model compatible with the Visitor pattern as was explained in the theory classes. Create a package `org.ibcn.gso.labo5.visitor` and add the following interfaces:

NoteElementVisitor

- `void visit(Note note)` : called once when visiting a `Note`.
- `void visit(Message message)` : called once when the content for the `Note` that is visited is of the type `Message`.
- `void visit(CheckList checkList)` : called once when the content for the `Note` that is visited is of the type `CheckList`.
- `void visit(CheckListItem checkListItem)` : called for each item that is present in the `CheckList` that is being visited.

NoteElement

- `void accept(NoteElementVisitor visitor)` : accepts a visitor for the element, allows you to then call the appropriate visit methods on the visitor (this is what we call double-dispatch).

Complete step 1 by implementing `NoteElement` for each class in the model except for the Credentials class of course.

Note on visitors: *we use the visitor pattern here so you can get some practical experience with it as it is good to know some of the concepts behind it. However, modern programming languages such as Java often have some kind of reflection capabilities (infrastructure that allows the program to see and manipulate itself) which provides a superior solution to the set of problems that can be solved with the visitor pattern.*

For example: as you have just experienced, the visitor pattern is intrusive as your model classes need to be adapted to be made compatible with the pattern. Reflection strategies can be implemented without requiring changes to the model classes. A good starting tutorial on reflection can be found here: http://tutorials.jenkov.com/java-reflection/index.html

Step 2: Implement the JSON visitor

Now that you've designed a visitor pattern and extended the model so it can accept visitors, a specific visitor implementation can be provided that converts `Note` objects to JSON (which is required for the HTTP request handling).

In the package org.ibcn.gso.labo5.json add a class `NoteToJson` that implements the interface `NoteElementVisitor`. In addition to the visit methods that need to be implemented for the interface, we'll also add:

- `NoteToJson(Note note)`: a constructor that takes on the `Note` object that needs to be converted.
- `JsonObject getJSON()`: returns a `JsonObject` that represents the JSON conversion for the given `Note` object.

Think carefully about how this visitor can be implemented. You can test your `NoteToJson` class by constructing a `Note` object that contains several entries (containing regular messages and checklists), converting it to JSON using `new NoteToJson(note).getJSON()` and then printing the result in the `Main` class.

Tip: If you're having trouble with the visitor pattern, please look at the animated slide and notes in the file VISITOR_PATROON.pptx on Minerva for a visual overview of the logic flow for converting to JSON using this approach.

Step 3: Implement the basic HTTP handlers

Now that you have a functional mechanism for converting `Note` objects to JSON, we can start working on the HTTP handlers that implement the server-side of the notebook application. The Spark Framework library allows us to easily create these handlers by implementing their `Route` interface which specifies the following method:

- `Object handle(Request req, Response resp)`: handles a HTTP request. The parameter `Request` gives access to request data such as HTTP headers, query parameters, etc… The `Response` parameter enables manipulating the HTTP response (status code, response headers, etc…). The return type Object is sent over the network as the body of the HTTP response.

Because Route only has one method, we can implement it using a lambda expression in Java 8. The following example uses an HTTP query parameter from the request to write a hello message to the response:

```
Route routeExample = (req, resp) -> {
    return "Hello " + req.queryParams("name");
};
```

We will now use Spark to set up a Chain of Responsibility with support for four HTTP operations:

Listing all notes

```
Spark.get("/api/notes", (req, res) -> {
    return null;
});
```

If the HTTP request is a GET request that matches the path "*/api/notes*" the above handling code should be triggered. Complete this code in the `Main` class by executing the following:

1. Get a reference to the `Storage` singleton instance (`org.ibcn.gso.labo5.storage.Storage`)
2. Retrieve the Storage Map for the current user using `getStorageForUser(Credentials user)`. As there is no authentication yet, you can just pass `null` as argument.
3. Iterate over each `Note` object in the storage `Map` collection to convert these to JSON Object and add these to a new JSON array.
4. Return the JSON array containing all `Note` objects as a String.

Getting a single note

```
Spark.get("/api/notes/:id", (req, res) -> {
    return null;
});
```

If the HTTP request is a GET request that matches the path "*/api/notes/:id*" where the parameter id represents the id of the `Note` to be retrieved, the above handling code should be triggered. (The colon at the start of `:id`, means it will be replaced by any value entered there.) Complete this code in the `Main` class by executing the following:

1. Get a reference to the `Storage` singleton instance and retrieve the storage `Map`.
2. Get the id of the requested note: `req.params(":id");`
3. Use this id to lookup the corresponding note in the storage `Map`.
4. Return the found note.

Creating a new note

```
Spark.post("/api/notes", "application/json", (req, res) -> {
   return null;
});
```

If the HTTP request is a POST request that matches the path "*/api/notes*" the above handling code should be triggered. Complete this code in the `Main` class by executing the following:

1. Read the body from the HTTP request (`req.getBody()`) as a JSON object using the `org.ibcn.utils.json.Json` class.
2. Convert the JSON object to a `Note` object using the provided `JsonToNote` class.
3. Generate a unique id for the new note: `UUID.randomUUID().toString()`
4. Get a reference to the `Storage` singleton instance and retrieve the storage `Map`.
5. Add the note converted from JSON to the `Map` collection (using `put`) with the generated id as key.
6. Return the created `Note` as a JSON String.

Updating an existing note

```
Spark.put("/api/notes/:id", "application/json", (req, res) -> {
    return null;
});
```

If the HTTP request is a PUT request that matches the path "*/api/notes/:id*" where the parameter id represents the id of the `Note` to be updated, the above handling code should be triggered. Complete this code in the `Main` class similarly to how creating a new note was implemented.

Deleting a note

```
Spark.delete("/api/notes/:id", (req, res) -> {
    return null;
});
```

If the HTTP request is a DELETE request that matches the path "*/api/notes/:id*" the above handling code should be triggered. Complete this code in `Main` class by using the `Map.remove()` method on the storage `Map` collection. Return the removed Note as a JSON String.

These typical operations are often referred to as CRUD (create, read, update and delete).

The Spark framework implements logic that takes care of the precedence of each handler in the chain. Each new HTTP request is always passed on to the most specific handler in the chain: e.g. sending a HTTP GET to *"/api/notes/2e961211-d04b-42aa-a30e-fb31817773fb"* matches the method and the path for the "List all notes" handler, but because of the additional id in the path, the request is forwarded to the "Getting a single note" handler.

When you start the application now, you can use your browser to go to the following address: http://localhost:4567

With the handlers you've implemented in this step, you should be able to use the UI to create, edit and delete new notes. Of course, there is no concept of users yet (everyone visiting the page sees the same notes) and when you restart your Java application all the created notes are gone. We will take care of this in step 4 and 5.

Step 4: Add authentication

To introduce the concept of users for the application, we will add authentication using a built-in feature of the HTTP specification called Basic Auth[1]. When the client application sends an HTTP request to the server for the first time, the server will respond with a special HTTP header in the response, indicating that authentication is required. When receiving this header, the browser will show a login dialog prompting for a username and password. A Base64 encoded version of the username and password combination is then sent along with each client request. If these so called credentials are correct, the server will continue operating as before in step 4 (and otherwise an authorization error will be thrown).

[1] In a real life application Basic Auth is never safe enough if not sent over an encrypted connection, using HTTPS (i.e. HTTP over SSL). In Basic Auth your credentials are sent to the server, so unencrypted connections could just be read by others on the network. For this lab however, this is not a concern.

The `Main` class already contains a `getAuth(Request request)` method that will help you with the authentication implementation. A static Credentials array `REGISTERED_USERS` is used to configure the available user accounts. In a real life application, this information would be encrypted and stored in some kind of database.

You will implement this additional authentication step by adding an additional handler (in the `Main` class) to the Chain of Responsibility of the Spark framework:

```
Spark.before((req, resp) -> {

});
```

This specific handler will be called at the beginning of the chain for all HTTP requests. Implement this handler to execute the following actions:

1. Try to retrieve the credentials using the `getAuth(Request request)` method that we've provided.
2. Check if the credentials are present (using the `Optional` object – see below for explanation)
3. If the credentials are not present:
    a. Set the special HTTP header in the response: `resp.header("WWW-Authenticate", "Basic realm=\"GSO Notes App\"");`
    This will trigger the browser to prompt for username/password.
    b. Interrupt the handling chain (we don't want to perform additional handling if the user is not authenticated): `Spark.halt(401);`
    401 is the HTTP status code that indicates that the client is not authorized to perform this request.

The provided `getAuth(Request request)` method inspects the request and tries to parse the Base64 encoded username/password combination in the HTTP request header "Authorization" (if this is provided by the client). If the credentials cannot be parsed, an empty instance of the `Optional` class is returned.

Now that we've enabled Authentication, you can complete step 4 by using `getAuth()` to retrieve the credentials in all your handlers so you can use the username of the Credentials instance as a parameter for the `getStorageForUser()` method instead of `null`. We will use this parameter in the final step to implement storage.

***Note** that the Chain of Responsibility pattern ensures us that these handlers can only be called with Credentials included in the request, else they would have been halted before reaching the handler. This means that it isn't necessary to check the Optional in this part of your code.*

Step 5: Implement the storage Proxy

To complete the application, you will extend the current `Storage` implementation with a dynamic Proxy that allows the Notes to be persisted on disk. Java has a built-in mechanism for these kind of proxies based around the `InvocationHandler` interface.

Using this mechanism, you will create a proxy for the `HashMap` instance that is currently returned in the `getStorageForUser` method of `Storage`. This proxy watches the methods that are being called on the original `HashMap` and makes sure that changes are propagated towards disk so that the Notes are persisted even when the application is restarted.

The implementation of the proxy consist of two parts: first you need to create a new class called `FileStorageMap` in the package `org.ibcn.gso.labo5.storage`, that implements the `java.lang.reflect.InvocationHandler` interface. `FileStorageMap` should have the following methods:

- `FileStorageMap(String location, Map<String, Note> proxiedMap)`: the constructor for the FileStorageMap that takes on two parameters. The `location` String is a reference to the storage folder to use, while the `proxiedMap` is the collection that is wrapped by the proxy. Initialize the `FileStorageMap` by checking if the folder exists (use `File.mkdirs()` if this is not the case) and then perform the following actions:
  - Get a list of all the files in the storage folder.
  - Read each file from disk as a String using the Apache Commons `FileUtils.readFileToString` method (encoding is UTF-8).
  - Parse the resulting Strings to JSON objects which can then be converted to `Note` objects using the `JsonToNote` class.
  - Add entries to the `proxiedMap` for each `Note`. The name from the originating file is the unique id that should be used.
- `Object invoke(Object proxy, Method method, Object[] args)`: this is the handling method for your proxy and it will be called for each method that is executed on the `proxiedMap`. Implement the following logic here:
  - Use `method.getName()` to determine which method was called.
  - If a method was called that adds a new `Note` or updates a `Note` object (e.g. `put`), use the `FileUtils.writeStringToFile` method to write the JSON String representation of the note to disk. Think about the signature of the method that is called to get a reference to the `Note`, e.g. if `put` is called the signature is `put(String key, Note value)`, so the `Note` is the second argument and can be retrieved as follows:
    ```
    Note note = (Note) args[1];
    ```

- If a method was called that deletes a `Note` (e.g. `remove`), get a reference to the `File` that represents the `Note` and remove it from disk.
- Conclude your implementation by invoking the original method on the `proxiedMap` and returning the result:
  `return method.invoke(proxiedMap, args);`

The second part of the proxy implementation is registering your `InvocationHandler` implementation as a new Proxy instance and using it in place of your original object. For the `Storage` implementation this means that we need to update the code for the `getStorageForUser` method. So instead of returning the same `HashMap` instance for each user like we are doing now:

```
private Map<String, Note> tmpStorage = new HashMap<>();

public Map<String, Note> getStorageForUser(Credentials credentials) {
    return tmpStorage;
}
```

We will create a dynamic Proxy based on your `InvocationHandler` implementation to return a wrapped `HashMap` instance that is backed by a folder on disk (containing json files for all the notes in the notebook):

```
public Map<String, Note> getStorageForUser(Credentials credentials) {
    String location = "./storage/" + credentials.getUsername();
    return (Map<String, Note>) Proxy.newProxyInstance(
            Map.class.getClassLoader(),
            new Class<?>[]{Map.class},
            new FileStorageMap(location, new HashMap<>())
    );
}
```

The first parameter of the `newProxyInstance` method specifies which classloader that should be used to load the Proxy. This is an internal Java mechanism that is out-of-scope for this course. In most cases it will suffice to use the classloader for the class that we're proxying (`Map` in this case).

The second parameter is an array of interfaces that this Proxy should pose as, by specifying a `Class` array containing `Map.class` here, we allow the Proxy being used as if it were a regular `Map` instance. Think of this as the "`implements Map`" part of you class definition, but expressed as a method parameter using a class array. The <?> wildcard is used because this array could contain multiple class types, e.g. `Class<Map>, Class<Serializable>, …`

The third and final parameter is the `FileStorageMap` that implements `InvocationHandler` and thus contains the logic for our Proxy.

If all steps were executed successfully, you should now have a working web-based notebook implementation.

Submitting your solution

1. Important: labs are solved individually
2. Bundle your source solution files in one zip file named ***lab5_name_surname.zip*** (e.g. *lab5_bruno_volckaert.zip*)
3. **All .java files** need to be included in the zip file!
4. **No UML class diagram** is required for this Lab Session, but feel free to create one if it can help you with your implementation.
5. Send this zip-file via Minerva dropbox to **Wannes Kerckhove, Thomas Dupont** and **Bruno Volckaert**, deadline for sending this is exactly one week from the start of this lab. I.e. for this lab the deadline is Friday 17th of November at 15:59.