Authentication Component Design Specification

Author : Simon Diemert
Created On : 2015-07-12
Updated On : 2015-07-17

Document Purpose

The purpose of this document is to specify the design for the stand-alone authentication component of Physician's Data Collaborative system. The design specified herein is intended to provide guidance for developers working on either the auth component, or a component intended to interact with the auth componet. This design is intended to meet a subset of the system requirements for authenticating users. The requirements that will be addressed are described below.

Related Documents

- This document may refer to the models and diagrams provided in the `PROJECT_ROOT/design/models/` directory of the code repository. StarUML was the modelling tool that was used to generate and maintain these documents, it can be downloaded from: http://staruml.io/.
- For convenience this document may be converted to a word document (or back to markdown) via Pandoc, a utility for converting between document types. See http://pandoc.org/.
- Much of this component's design is driven by the PDC network's Security Requirements Specification, this can be found on the project Polarian

instance. This requires login credentials. A brief summary of the requirements for this component are given below.

Summary of Requirements

These requirements are derived from general system requirements, which can be found in the PDC project Polarian instance. Specific documents of interest include the Security Requirements Specification. This section does not give a comprehensive review of requirements, other documents may need to be consulted for a more detailed explanation.

Auditing

The system shall provide a means of tracking user activity (within the authentication) component. Each action a user takes shall be recorded as an event, the following information **must** be recorded:

- date and time
- event type or description
- user identification (when applicable), such that the true identification of the user can be determined (either via manual or automated process)
    - username
    - jurisdiction
    - role (if applicable)
    - group (if applicable)
- event outcome (when applicable/possible)

Specific events of interest are:

- All authentication events, both successful and failed.
- Verification of a previous authentication, both successful and failed.
- All events with respect to group or role of a user
- All user management events (addition, deletion, modification)

Users of the system shall not be able to access, view, modify, or delete the audit

records of the system.

Sensitive data, such as passwords, shall hashed or encrypted (such that the original cannot be determined) prior to being recorded in audit logs.

Audit logs shall be stored in a location where they can be backed up and accessed after the fact.

Access Control

The access control policies described in the Security Requirements Specification will be *partially* met by this design. Full group based access control is beyond the current scope for this work.

The system shall provide two roles:

- `user` which allows for basic authentication.
- `administrator` which allows one to control access for other users.

User Management and Restrictions

- The system shall allow *only* system administrators to view, modify, delete, update, etc... the authentication information for users.
- The system shall provide a simple user interface for system administrators to manage users and their roles.

User Authentication

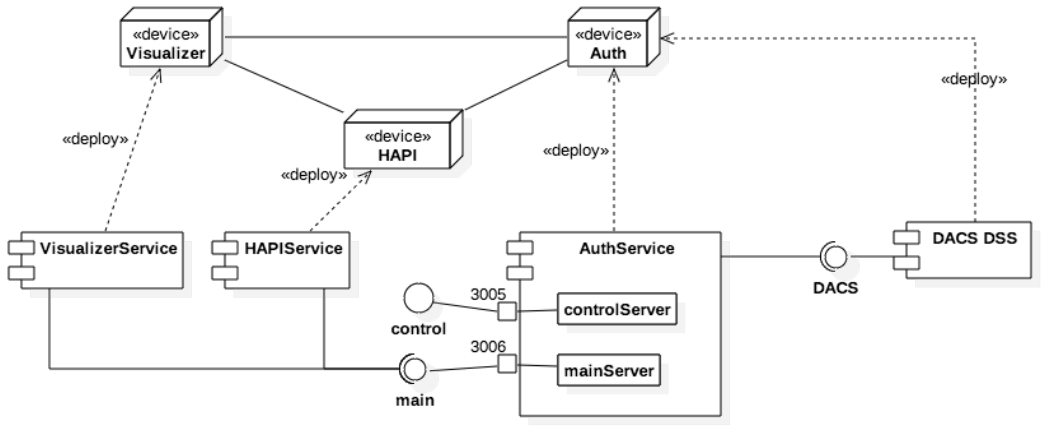
- The system shall provide a means of authenticating users against a list of allowed users.
- The system shall be able to verify that a previous authentication event occurred given an token.
- The system shall be able to associated roles with specific users and return the role information upon request

Summary of System

This authentication component (referred to as the auth component) is to be an HTTPS interface between the rest of the PDC system, and the DACS DSS user management system (https://dacs.dss.ca/). The persistence and management of user information is handled by a DACS DSS programming running on the same host as this authentication component.

The auth component consists of two web servers running on the same NodeJS process, but listening on different ports. One server (referred to as "control", on port 3006) provides a login and for system administrators to manage users. The other provides access to a REST API for authentication (referred to as "main" on port 3005).

The following diagram gives a high level view of the auth component deployment. Note that communication with the DACS component must occur via calls to the unix command line. All networked communication in the diagram is HTTPS (unless otherwise specified).



Code Structure

The structure of the code for this design is follows:

```
src/
    controller/
    model/
    view/
    util/
    middleware/
test/
    controller/
    model/
    view/
    util/
    middleware/
public/
init.js
package.son
```

This code structure follows a simple MVC paradigm. Controllers and boilerplate HTTP networking code are provided by the NodeJS Express framework: http://expressjs.com/4x/api.html#express. Views are HTML pages facilitated by HandleBars JS http://handlebarsjs.com/.

Design Concepts

Significant effort on the design of this software has been spent on ensuring that the external dependencies can be removed. As such, several classes have been created to provide a clean interface between:

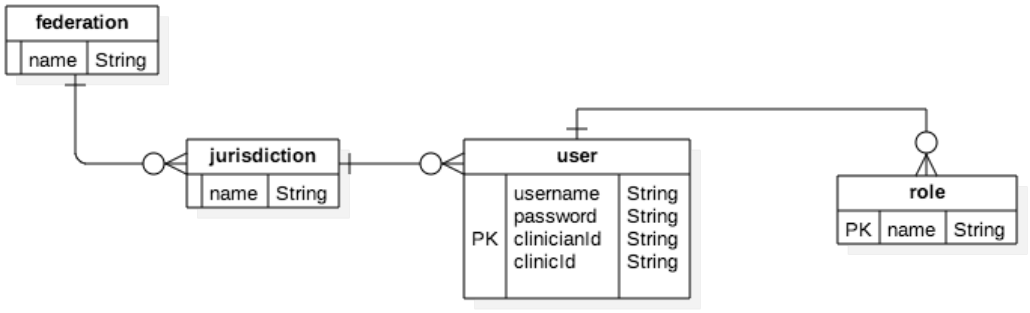
- Auth business logic and DACS
- Node Express server and auth business logic

These abstractions will likely improve the maintainability of the system should an external dependency change, need to be adjusted or, need to be replaced.

Data Model

This section presents a data model that is utilized by the auth component. This

model has many parallels with the DACS data model.

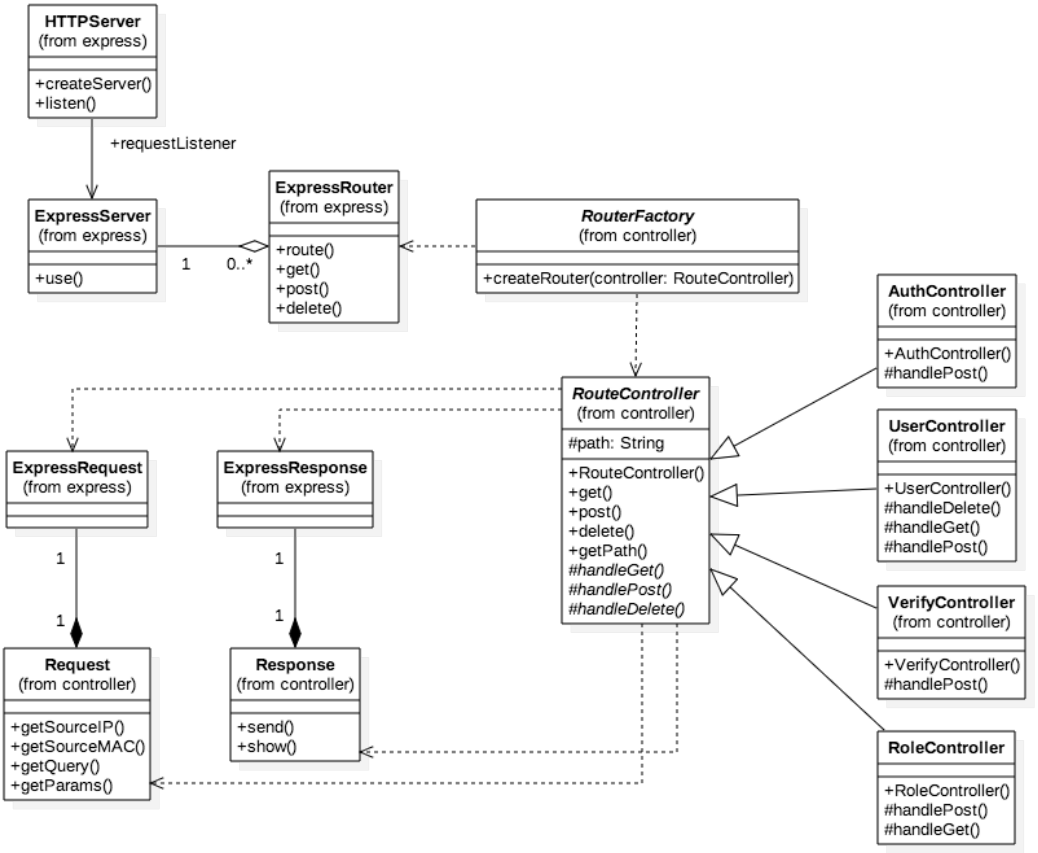


Class Design

The following provides a description of the class (formally object in JavaScript) structure for the auth component code base. Several diagrams are used to illustrate key relationships. In order to save space in the diagrams, operation signatures have been removed unless they are absolutely required to illustrate a concept; for detailed operation signatures, consult the UML model.

Interface with Express

This section describes the relationships between the Node Express web service framework and the auth application code. As previously mentioned, the design has attempted to mitigate the coupling between classes (objects) with the Express framework and the rest of the application.



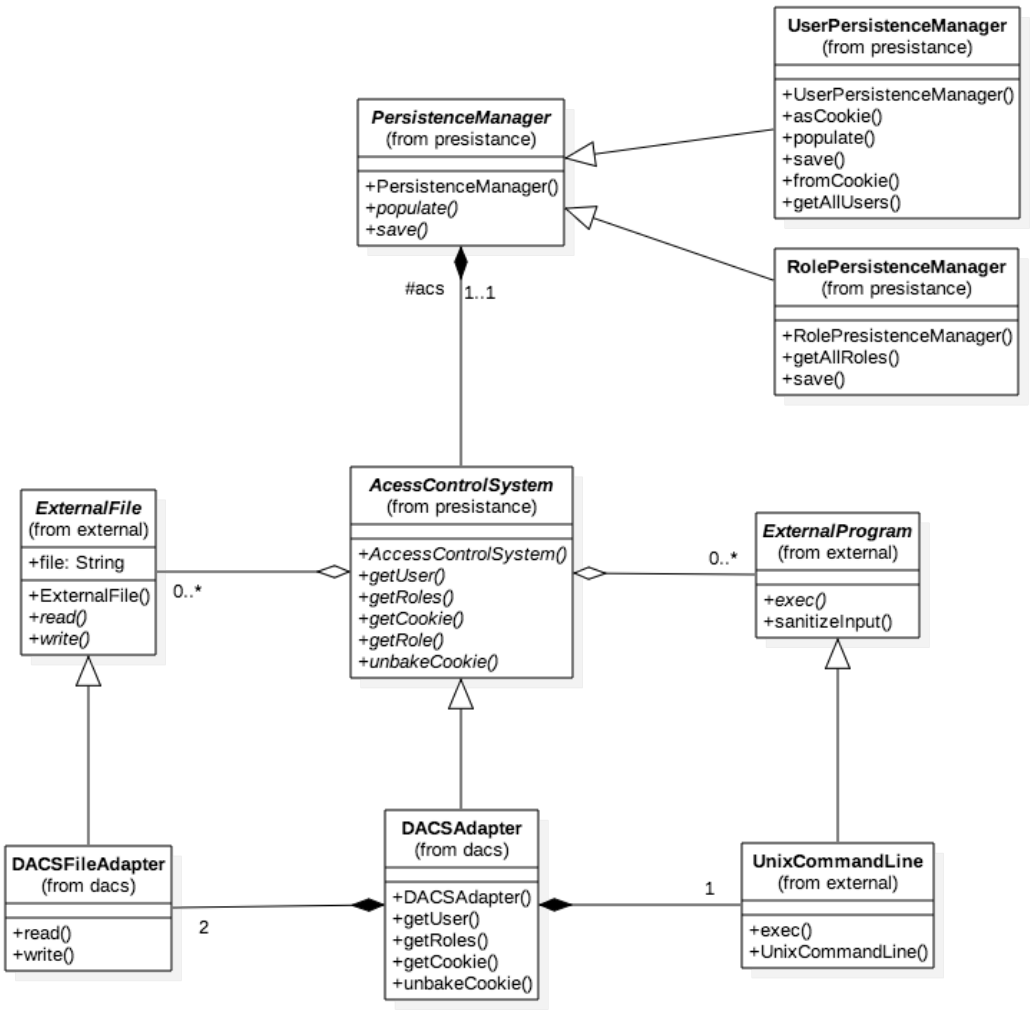
The UML class diagram above shows the use of the factory pattern to create Node Express *ExpressRouter* objects. The *RouterFactory* consumes a *RouteController* object and generates an *ExpressRouter*, which can then be provided directly to the *ExpressServer*.

The *RouteContoller*, *Request*, and *Response* objects are the only points of contact with the Node Express framework. Each request is serviced by the bound *RouterController* and accesses any information regarding requests or responses through the *Request* and *Response* objects, which will return the data (if applicable) from the *ExpressRequest* and *ExpressResponse* objects.

Interface with DACS

The section describes the relationships between the DACS Distributed Access Control System and the auth application code. This interface was created to

reduce the amount of coupling between the DACS system and the application code. It uses a facade design pattern where one interface, the *AccessControlSystem*, defines the functionality that is available to the rest of the system.



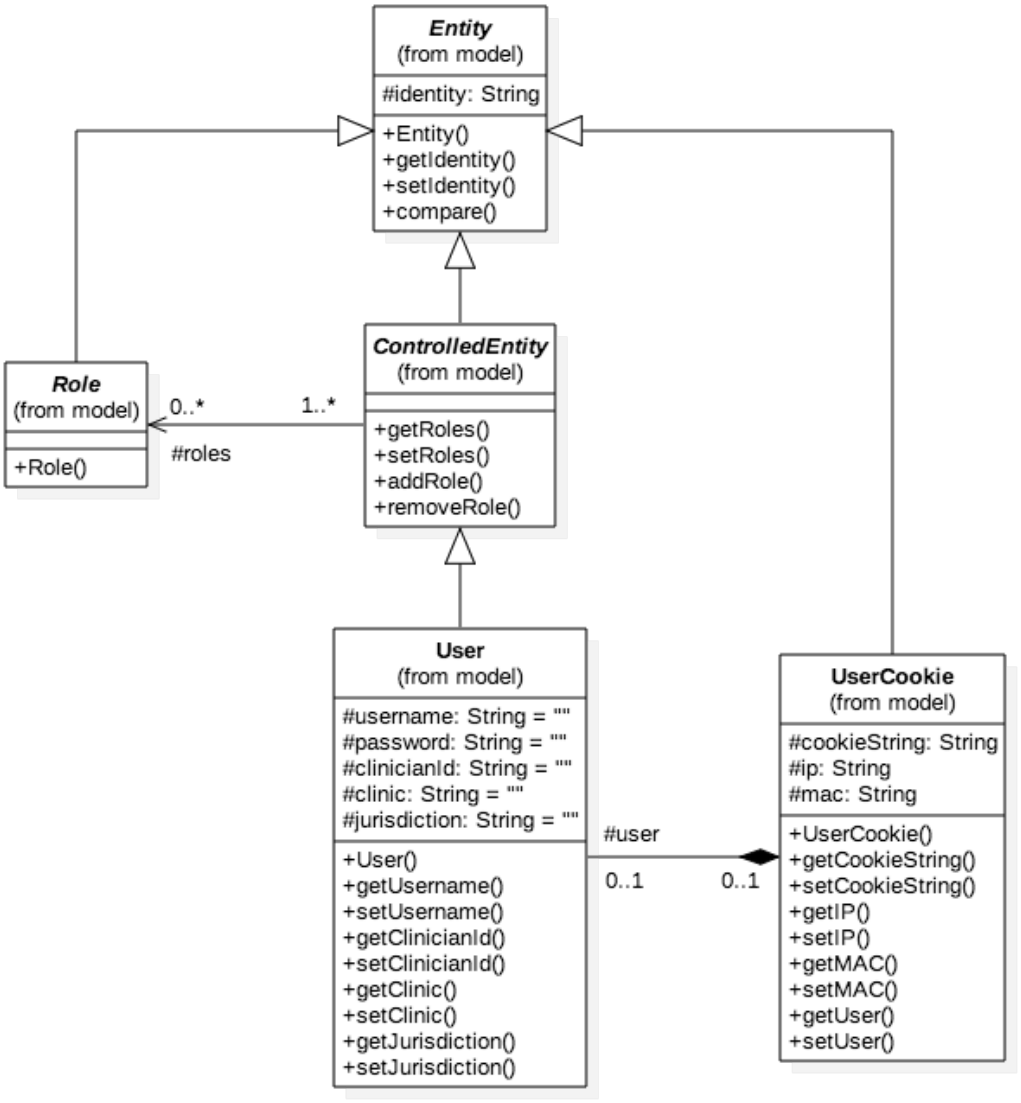
Behind the *AccessControlSystem* facade is an adapter for specifically for DACS (this could be swapped out for another adapter if desired). It must access DACS through both external programs (cookies and private data via the command line) and through the file system (for roles and users). Helper classes are provided for both of those. These classes consume and return business objects that are used by the rest of the application code base.

The *PersistenceManager* class is the interface with the rest of the auth application code. It, and its sub-classes, are the only way to make requests to the *AccessControlSystem*. This extra layer of abstraction means one can swap out the *AccessControlSystem* at any point and provide an entirely different way of storing user data.

Model Classes

This section describes the relationships between the classes (and objects) that represent the data model. All model objects inherit from an *Entity* object. This provides a bare minimum functionality for each object, including an identity attribute. A *ControlledEntity* is an object that represents an entity in the real world that has its access controlled, for example a *User* of the system has its access controlled by the auth component. However, a *Role* does not have its access controlled, instead it governs kind of access that an entity has.

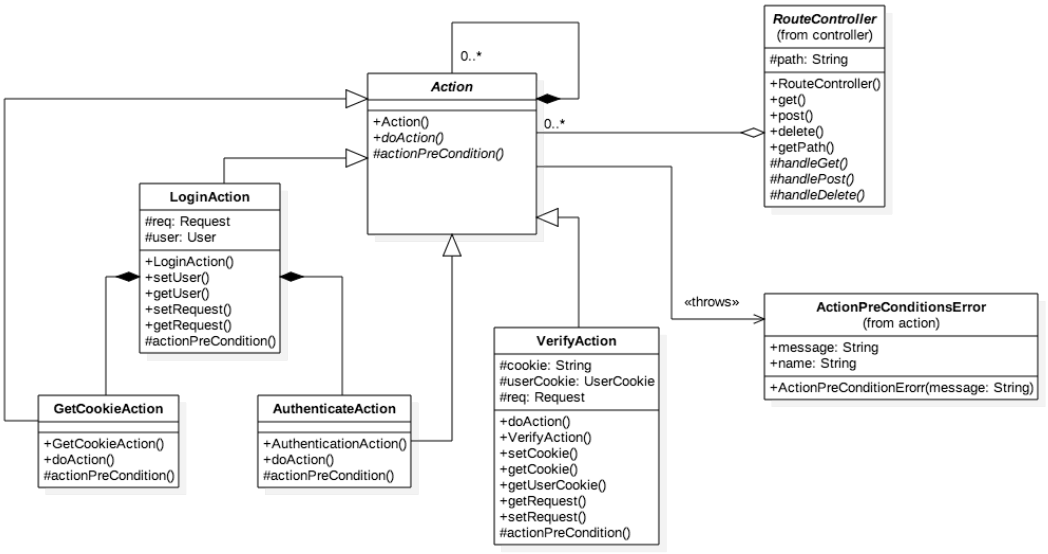
The *UserCookie* object is a wrapper around a *User* object that supports the concept of a cookie, or encrypted representation of the *User* object. Both the *User* and *UserCookie* objects are used to interact with the *AccessControlSystem* interface.



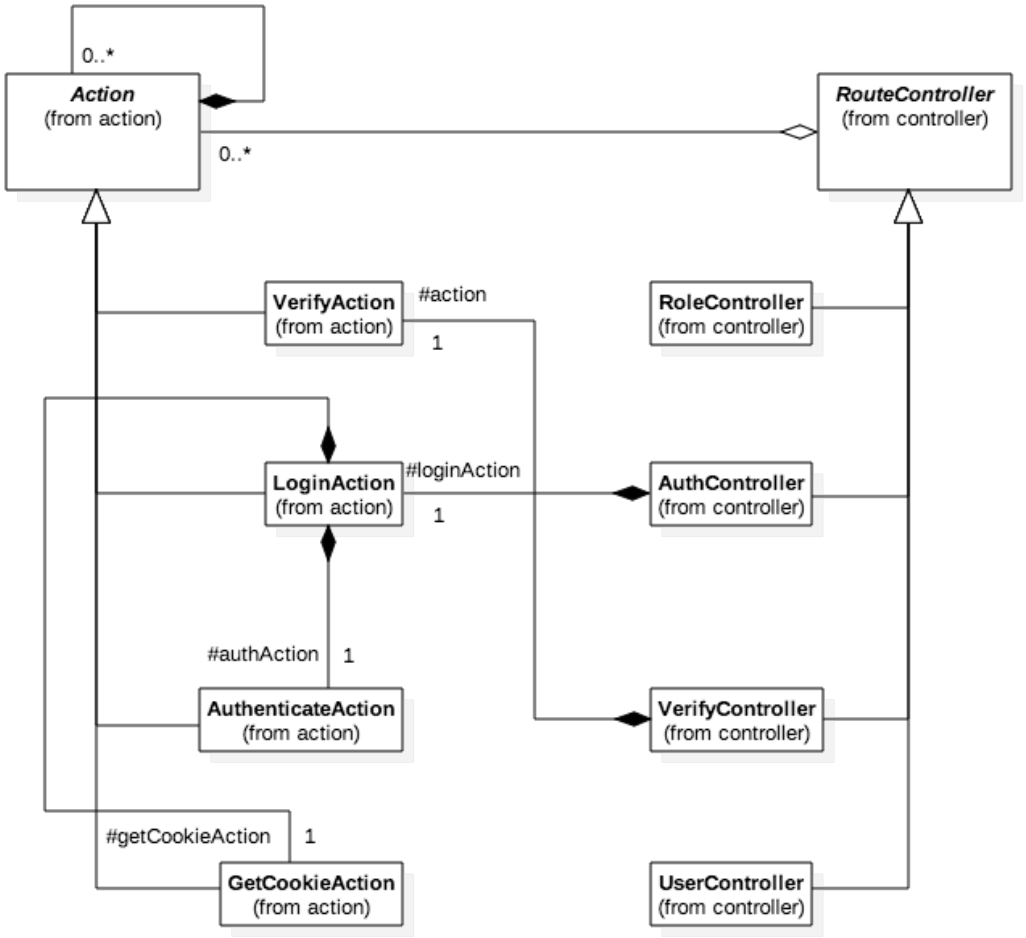
Actions

The *Action* classes provide a means of defining workflows or processes within the application. *Action*s are utilized by controllers to service requests. The relationships between actions is a modification of a traditional strategy pattern. As the UML class diagram below indicates, *Action*s may include other *Action*s and may call them as needed. All *Action*s must provide a `doAction()` method that triggers the main process encoded by the *Action*. Prior to executing it's main

workflow, each *Action* should run its `actionPreCondition()` method to determine whether the necessary structure exists for it to execute successfully. If the *Action* cannot execute successfully, it should thrown an *ActionPreConditionError*.

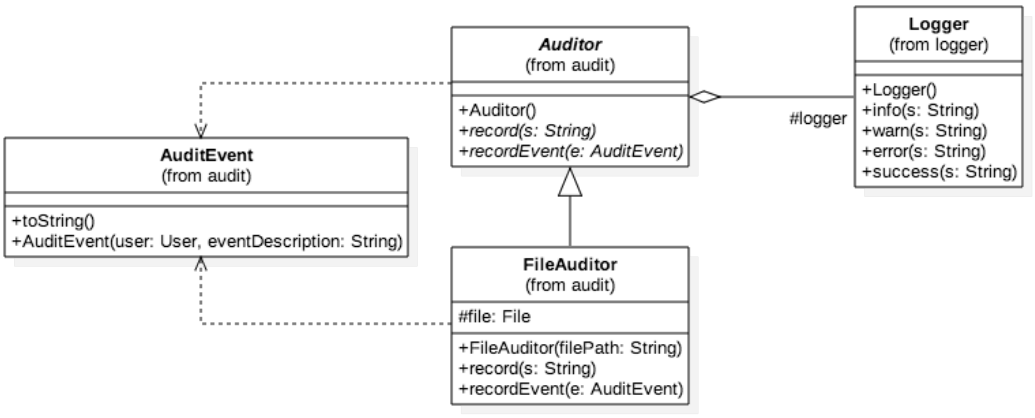


The next diagram shows the relationship between *Action*s and the *RouteController*s.



Auditing

The *Auditor* class provides an interface for auditing components. It, coupled with an *AuditEvent* provide a means of auditing key interactions within the system. The *Auditor* class can be extended to provide an interface to several different auditing frameworks. The default class is *FileAuditor* which simply writes events to file. *Auditor*s also utilize a *Logger* singleton to write their output to standard out.

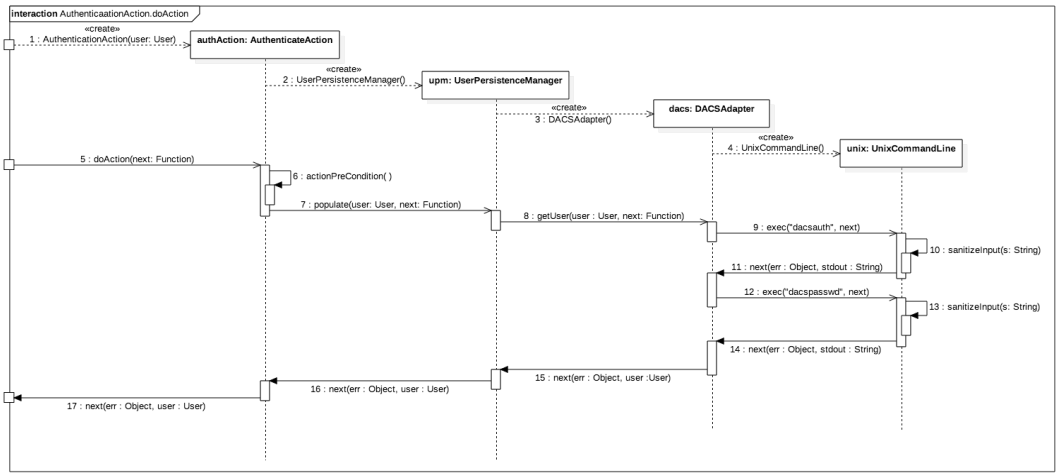


Interactions

This section describes key interactions between different objects in the auth application. Many of these interactions correspond directly to actions that are described by an *Action* object.

Authentication

Authentication of a *User* is preformed by constructing a *User* object and passing it to the *AccessControlSystem* (a *DACSAdapter* in this case) via the *UserPersistenceManager*. The *AccessControlSystem* will validate the *User* and return (asynchronously via callbacks) a populated *User* object or the `err` parameter of the callback will be set; indicating that the authentication failed.

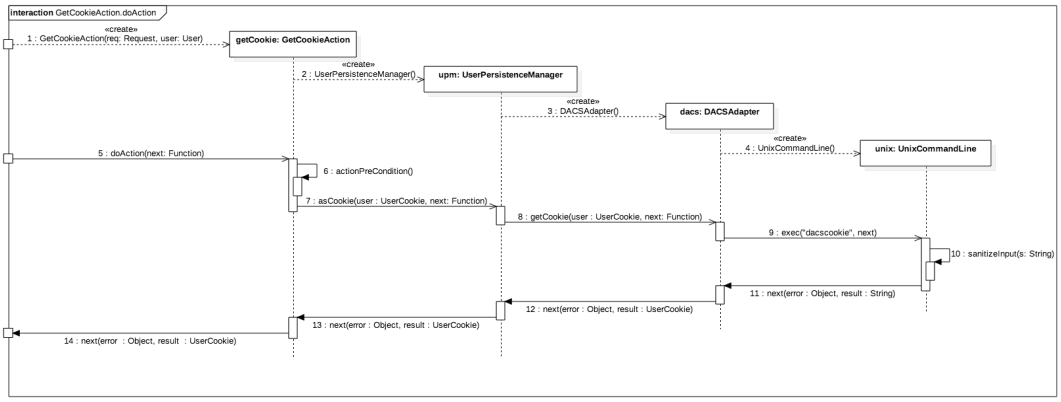


Cookie Management

This section describes *Action*s that are related to generation and decryption of cookies. There are two such *Action*s, *GetCookieAction* and *VerifyAction*.

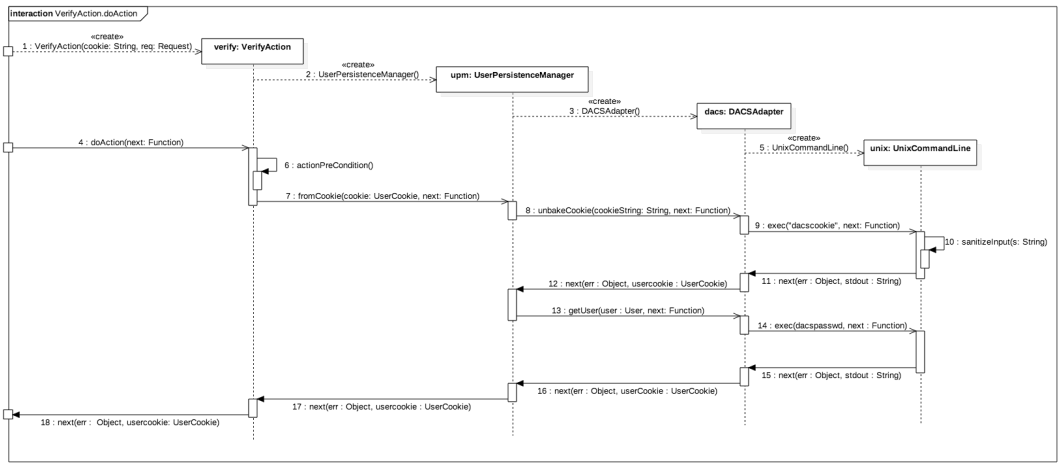
GetCookie

The *GetCookieAction* obtains a cookie from the *AccessControlSystem* that represents the provided *User* object. A *UserCookie* object is returned (via asynchronous callback) to the parent of the *GetCookieAction*. The following UML sequence diagram describes interaction.



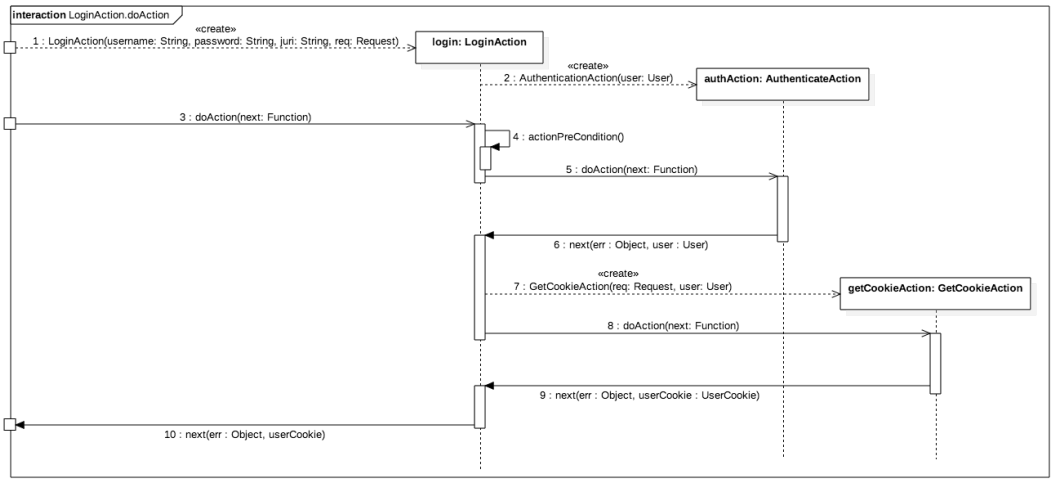
Verify

The *VerifyAction* verifies that a cookie is valid. The action "unbakes" or decrypts a cookie that was previously generated by the *AccessControlSystem* and returns a *UserCookie* (via asynchronous callback) object to the parent of the *VerifyAction*. If the cookie could not be verified, due to expiration or being invalid, an error will be returned. the The following UML sequence diagram describes interaction.



Login

The *LoginAction* allows users to log into the auth component. This action requires a takes in username, password, jurisdiction and returns a fully constructed UserCookie object or fails with an err. The response is provided by asynchronous callback. This action uses two other actions, namely *GetCookieAction* and *AuthenticationAction*. The following UML sequence diagram shows the interaction with these two other actions.



API

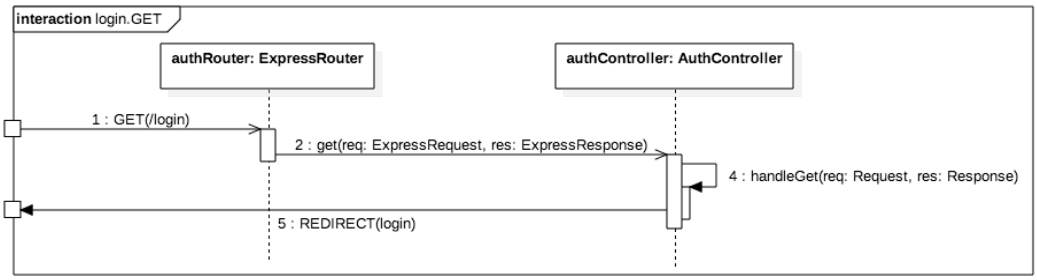
This section describes interactions that are meant to service API routes. See the API section for details about routes. The diagrams presented here show only the *normal* case of operation, that is where the request is successful. Failure cases (500, 404, 401, 400, etc.) must also be implemented. The following sub sections are organized by the route.

/login

Describes the interaction required for servicing the `/login` route. This route can service HTTP POST and GET requests.

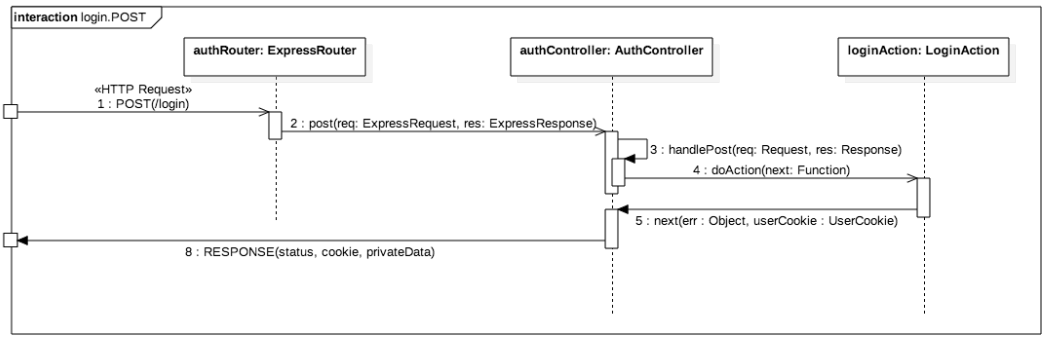
GET

HTTP GET requests are serviced by simply redirecting the client to the login page of the application.



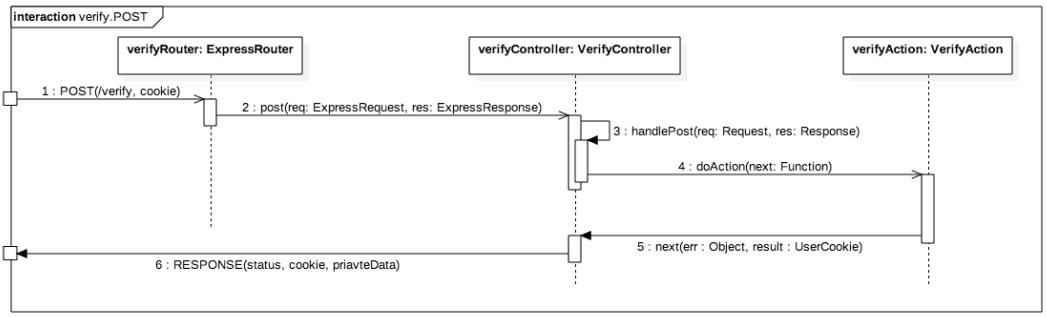
POST

HTTP POST requests are serviced by executing the *LoginAction* workflow. The result of which is a cookie and user data to be returned to the client.



/verify

Describes the interaction required for verifying an existing cookie. This route can service **only** HTTP **POST** request.



API

The following section describes the client facing Application Programming Interface (API). It provides several routes for logging into the system, validating an existing login, and user management.

/auth/login

Allows users to login to the application and obtain a cookie for later use.

GET

Responds by redirecting the client to a login screen. User is not authenticate, no cookie issued.

POST

This route will generate a cookie that can be used to track a user through the system. Requires the following items be embedded in request json body:

- username as: `user`
- password as: `pass`
- jurisdiction as : `juri`
- respond as : `respond`

If login is successful, the response will be a JSON string of the following format:

- `{ "cookie" : "COOKIE_STRING" , "message" : "SOME MESSAGE" }`

The response status code will be one of:

- `200` - Authentication completed successfully
- `400` - Request was not well formed (see above)
- `401` - The user could not be authenticated due to invalid credentials.
- `500` - An error occurred in the authentication process.

In the event of a failed authentication (status code `500` , `400` , or `401`) the `cookie` field of the response will be `null` and the `message` field will contain an error description.

If the authentication is successful (status code `200`) the `cookie` field of the response will be a string that is the cookie for the authenticated user. The message field will contain a success message.

/verify

This route allows users to verify an existing cookie and obtain the private data associated with the cookie.

POST

This route verifies that an existing cookie is valid. This route requires that the cookie be a string embedded in the body of the post request. Specifically,cookie string must be accessible via: `request.body.bakedCookie` .

This route will return the following format:

- `{ "data" : DATA_OBJECT, "message" : "SOME MESSAGE" }`

The `data` field of the response will contain an object that has user information which can be used to identify the user later:

- `user.clinic`
- `user.clinician` '

The response status code will be one of:

- `200` - Verification completed successfully
- `400` - Request for verification was not well formed, i.e. there was not `request.body.bakedCookie` field
- `401` - Verification failed, this means the cookie now invalid. The cookie may be expired or have been modified/tampered with.
- `500` - Verification failed due to an error during verification.