DE4A
DIGITAL EUROPE FOR ALL

# Guidelines for the VC pattern flow simplification

| Document Identification | | | |
|---|---|---|---|
| Status | Final | Due Date | 06/07/2022 |
| Version | 1.0 | Submission Date | N/A |

| | | | |
|---|---|---|---|
| Related WP | WP5 | Document Reference | |
| Related Deliverable(s) | | Dissemination Level (*) | PU |
| Lead Participant | UM | Lead Author | Martina Šestak (UM) |
| Contributors | | Reviewers | N/A |
| | | | N/A |

**Keywords:**

VC pattern, WebSockets, Evidence/eProcedure Portal.

# Table of Contents

# List of Figures

# List of Acronyms

| Abbreviation / acronym | Description |
|---|---|
| DE4A | Digital Europe for All |
| VC | Verifiable Credential |
| VP | Verifiable Presentation |
| SSI | Self-Sovereign Identity |
| WS | WebSockets |
| STOMP | Simple Text Orientated Messaging Protocol |
| JS | JavaScript |
| AA | Authority Agent |

# 1 Introduction

## 1.1 Purpose of the document

This document describes the implementation steps necessary to simplify the VC pattern flow in the 2nd pilot iteration. The simplification refers to reducing the number of steps in the Student-Portal interaction when issuing or submitting Verifiable Credentials.

These guidelines are limited to the audience of the DE4A Studying Abroad pilot.

## 1.2 Structure of the document

This document is divided into 3 main chapters:

- Chapter 2 – description of the simplified VC pattern interaction flows
- Chapter 3 - technical prerequisites that must be met to simplify VC pattern flow
- Chapter 4 – implementation steps, which must be taken, to change the Evidence/eProcedure Portals for 2nd pilot iteration

# 2 VC pattern simplified interaction flows

In the 2nd pilot iteration, the interaction flow between the Student and the Issuer's Evidence Portal includes the following steps:

1. The Student authenticates against the Portal by using his eIDAS credentials.
2. The Portal checks if there is an already established DID connection between the Portal and the Student's mobile phone. If true, the Portal offers the Student to use the existing connection.
3. If false, the Student proceeds to generate a QR code for establishing a new DID connection between the Portal and his mobile phone. The Student accepts the invitation displayed on his mobile phone.
4. The Portal automatically updates the interaction status and displays the 'Send VC offer' button, which enables the Student to proceed with issuing the VC. If the Student has more diplomas handled by the Data Provider, the Portal allows the Student to select which diploma he wants to be issued. The Student previews the credential data on his mobile phone and accepts the offer.
5. The Portal automatically updates the interaction status indicating that the VC has been sent and the Student needs to accept it. The Student accepts and stores the VC.
6. The Portal automatically updates the displayed interaction status (VC is accepted).

On the Verifier's eProcedure Portal, the interaction flow with the Student includes the following steps:

1. The Student authenticates against the Portal by using his eIDAS credentials.
2. The Portal checks if there is an already established DID connection between the Portal and the Student's mobile phone. If true, the Portal offers the Student to use the existing connection.
3. If false, the Student proceeds to generate a QR code for establishing a new DID connection between the Portal and his mobile phone. The Student accepts the invitation displayed on his mobile phone.
4. The Portal automatically updates the interaction status and displays the 'Send VP request' button, which enables the Student to proceed with submitting the VP. The Student select the VC on his mobile phone and submits it as a VP.
5. The Portal automatically updates the displayed interaction status (VP is valid or not) and the detailed results of the submitted VP validation.

# 3 Implementation prerequisites

The following changes need to be made in the implementation and deployment of the SSI Authority Agent (AA) and Evidence/eProcedure Portal:

1. The Authority Agent REST API includes the */webhook* endpoint, which registers any changes in the status of the interaction between two HL Aries agents (issuer's/verifier's and student's agent). The endpoint processes incoming events registered when a student accepts a DID connection invitation, accepts a VC offer and accepts a VP submission request (i.e. submits VP);
2. The Evidence/eProcedure Portal needs to include an additional POST API method, which will act as a listener to any status events received from the Authority Agent;
3. The Evidence/eProcedure Portal needs to implement a realtime mechanism for refreshing the view (screen) displayed to the student depending on the status changes. Depending on the already existing Portal implementation, this can be either *Single Page Application (SPA), WebSockets* or something else.

The following chapter will describe the **WebSockets** implementation in the Portal's backend, which will simplify the VC pattern by automatically refreshing the Portal frontend depending on the event notifications received from the Authority Agent. For the implementation examples, the Portal's backend is implemented in Spring Boot framework and the Portal's frontend is handled by the Thymeleaf templating engine.

# 4 SSI Authority Agent + Portal communication via WebSockets

The following sections describe the subsequent steps to be implemented in order to provide communication between the SSI Authority Agent and the Portal via the WebSockets protocol.

In iteration 2, the SSI Authority Agent is enriched with an „active“ component, which allows it to notify the Portal about any DID connection, VC issuance and VP submission changes. The communication between the Authority Agent and the Portal is achieved by using WebSockets. The high-level details of the communication flow are presented in Figure 1.

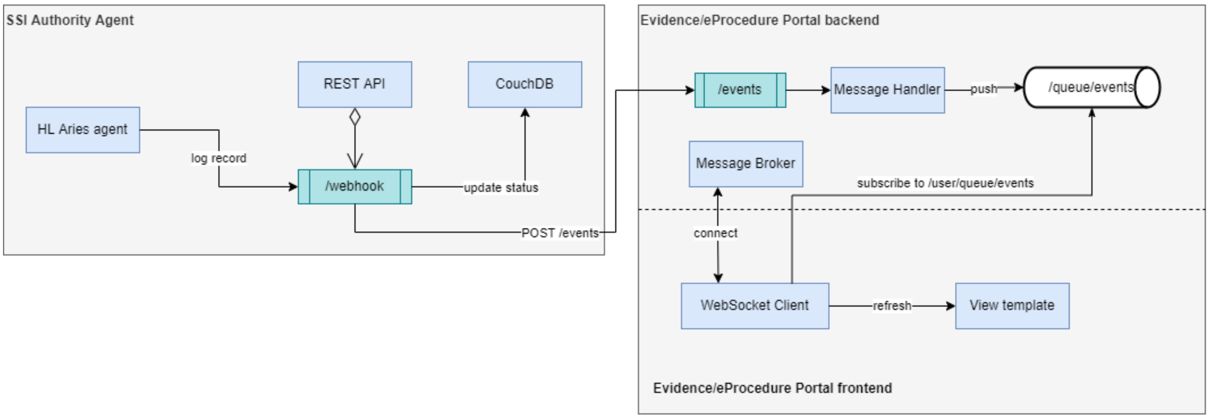

Figure 1. Communication flow between the SSI Authority Agent and Portal via WebSockets.

The AA REST API includes an integrated */webhook* endpoint, which listens to new log entries of the HL Aries agent (note: the endpoint is automatically included in the AA deployment and the HL Aries is configured to send notifications to this endpoint via the ARIESD_WEBHOOK_URL property, so no changes are required by partners besides updating the *docker-compose* file). The */webhook* endpoint filters the received log entries for updates in DID connection, VC issuance and VP submission responses to detect changes in the status of the interaction. Once it detects a change, it automatically updates the relevant status (DID connection, VC or VP status) in the CouchDB database and pushes a new event message to notify the Portal about this change. This is done by sending a POST request to the dedicated Portal's REST API method. **Important** prerequisite: the controller method on the Portal's side must enable external calls for unauthorized users as well – otherwise, the Portal's API method will be unreachable to the Authority Agent and no notification will be received on the Portal's side (e.g. in Spring Boot, this can be enabled by adding the `@PreAuthorize("hasAuthority('ROLE_ADMIN')")` annotation to the controller method). In the request body, the SSI Authority Agent sends a JSON object with the following structure:

```
public String protocol; -- “did-exchange” / “vcstatus” / “vpstatus”
public String userID; -- user identifier obtained from eIDAS login
public String uniqueIdentifier; -- connectionID (did-exchange) / piid (vcstatus, vpstatus)
public int statusCode; -- status of the interaction
```

The event object contains information about the *protocol* for which the status change has been detected (possible attribute values: *did-exchange*, *vcstatus* or *vpstatus*), the *user ID*, the *unique identifier* of the user's interaction (connectionID for establishing DID connection, PIID for VC/VP interaction flows) and the current *status code* of the interaction (e.g. -1 if no invitation has been generated). The status codes of the interaction correspond to the response values returned by */did-conn-status*, */check-offer-vc-status* and */check-request-vp-response* methods in the Swagger YAML file for the AA REST API (provided in the Github documentation for the SSI Authority Agent).

## 4.1 Portal: Configuring MessageBroker for handling WS messages

To support the simplified VC pattern flow, the Portal requires an implementation of a WebSocket Server i.e. a Message Broker component, which enables WebSocket (WS) message handling. In the Portal's backend implementation, the necessary dependencies should be added, which enable the usage of existing implementations for a WebSocket message exchange (e.g. *SockJS* and *spring-boot-starter-websocket* in Spring Boot). After these dependencies are included, a new class should be added, which includes a method for configuring a message broker. The configuration includes specifying message destination prefix(es) to which the WS client will subscribe later to receive messages. It is important to specify „queue" as the destination prefix, as this ensures that the WS message is sent only to a specific user session and not to all users.

Additionally, the STOMP protocol can be used inside the WebSocket implementation to provide support for an interactive Portal-Student communication. We enable STOMP by registering a separate endpoint (*/notifications* in the below example), which the WS client will use to connect to STOMP and read the messages.

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void configureMessageBroker(final MessageBrokerRegistry registry) {
        registry.enableSimpleBroker("/topic/", "/queue/");
        registry.setApplicationDestinationPrefixes("/swns");
    }

    @Override
    public void registerStompEndpoints(final StompEndpointRegistry registry) {
        registry.addEndpoint("/notifications")
                .withSockJS();
    }
}
```

## 4.2 Portal: Implementing MessageHandler methods

The MessageHandler component is the central implementation point of methods handling the WebSocket communication flows. In the code snippet below, the MessageHandler functionalities are implemented as a service called *NotificationDispatcher*. The implementation includes the methods for adding or removing a user's session to/from the list of sessions stored on the application level (*add()* method automatically create a new session for the user when he opens a web page and *sessionDisconnectHandler()* method, which automatically clears the session when the web page is closed).

The main method in the implementation is the *dispatch()* method, which takes care that the input event STOMP message is converted into a proper format (based on the MessageBroker configuration declared earlier) and pushed into the */queue/event* to be sent to the session listener.

```
@Service
public class NotificationDispatcher {
    private static final Logger LOGGER =
LoggerFactory.getLogger(NotificationDispatcher.class);

    private final SimpMessagingTemplate template;

    public static Set<String> listeners = new HashSet<>();

    public NotificationDispatcher(SimpMessagingTemplate template) {
        this.template = template;
    }

    public void add(String sessionId) {
        listeners.add(sessionId);
    }

    public void remove(String sessionId) {
        listeners.remove(sessionId);
    }


    public void dispatch(String eventData) {
        Gson gson = new Gson();
        SocketEvent event = gson.fromJson(eventData, SocketEvent.class);

        for (String listener : listeners) {

                SimpMessageHeaderAccessor headerAccessor =
SimpMessageHeaderAccessor.create(SimpMessageType.MESSAGE);
                headerAccessor.setSessionId(listener);
                headerAccessor.setLeaveMutable(true);

                template.convertAndSendToUser(
                        listener, "/queue/event",
                        new Notification(eventData),
                        headerAccessor.getMessageHeaders());

                LOGGER.info("Sent notification to " + listener);

        }
    }

    @EventListener
    public void sessionDisconnectionHandler(SessionDisconnectEvent event) {
        String sessionId = event.getSessionId();
        LOGGER.info("Disconnecting " + sessionId + "!");
        remove(sessionId);
    }
}
```

## 4.3 Portal: Implementing REST controller for receiving event messages

The */events* endpoint in the Portal's backend is a publicly assessible controller method, which forwards the received JSON object in the request body to the MessageHandler component, which will push the object to the */user/queue/event* to be sent to the user for his unique WebSocket client session.

The sample code snippet presented below is a sample implementation of the controller annotated as a *@RestController* written in the Spring Boot framework. Note that the *notificationDispatcher* corresponds to an injected instance of the MessageHandler component:

```
@PreAuthorize("hasAuthority('ROLE_ADMIN')")
@PostMapping(value = "/events")
public void receiveNotifications(@RequestBody String eventData){
    notificationDispatcher.dispatch(eventData);
}
```

## 4.4 Portal: WS Client subscribing to STOMP messages and updating views

As the final step, the frontend view templates (HTML files in most cases) need to be supplemented with the JavaScript code to implement a WS Client, which connects to the STOMP endpoint declared earlier. After adding the necessary JS libraries (SockJS, jQuery, STOMP websocket), the following code snippet needs to be added into the HTML file representing the view:

```
<script>
    $(document).ready(function () {

        let stompClient;

        if (!stompClient) {
            const socket = new SockJS("/notifications");
            stompClient = Stomp.over(socket);
            stompClient.connect({}, function () {
                stompClient.subscribe('/user/queue/event', function (response) {
                         // trigger action (e.g. refresh page)
                });
                stompClient.send("/swns/start", {});
            });
        }

    });
</script>
```

This code snippet will create a WS client listening to the */notifications* STOMP endpoint where WS messages are sent and subscribe it to the */user/queue/event* endpoint to receive any status messages relevant for the current user's session. The *subscribe()* method call includes a function, which handles the response, i.e. the received message with the status change information, so the implementation varies depending on the action needed to handle the status change (e.g. trigger refreshing the page, additional checks or something else). In case that the Portal is implemented as a Single Page Application, it is enough to include this code only once – otherwise, the WS client needs to subscribe to the STOMP endpoint on each web page.