WATCHME

Workplace-based e-Assessment Technology for Competency-based Higher Multi-professional Education

Deliverable 5.1: Description of integration of systems and components

Delivery month Annex I	10		
Actual delivery month	10		
Lead participant: Jayway	Work package: 5	Nature: R	Dissemination level: PU
Version: 1.0			

Project coordinator

Dr. Marieke van der Schaaf

Utrecht University
Faculty of Social and Behavioral Science
Department of Education
PO Box 80.140
3508TC Utrecht
The Netherlands

Telephone: +31 (0)30 253 4944
Email: M.F.vanderSchaaf@uu.nl

This project has received funding from the European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreement no 619349.

Table of contents

1. Executive Summary

This deliverable describes the proposed architecture for integrating the just-in-time feedback (JIT) and visualisation (VIZ) modules within the overall WATCHME system landscape, the major actors of which are outlined in Figure 1 below:

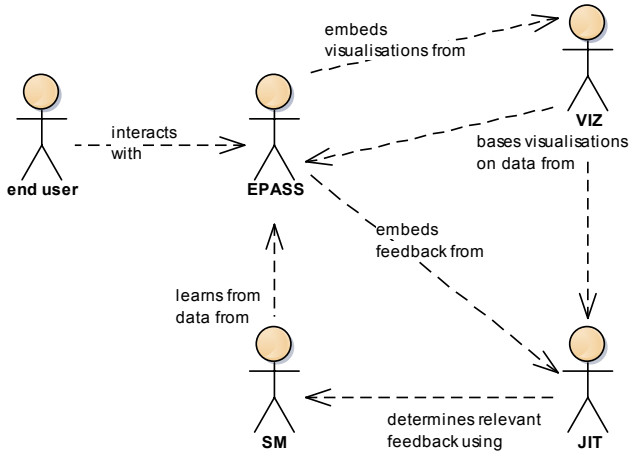

Figure 1: Overview of WATCHME system landscape

Specifically, this deliverable presents:

- The JIT/VIZ interaction with the student model (SM)
- The JIT/VIZ interaction with the electronic portfolio and assessment support system (EPASS)

The intended audience of this document are software architects and developers wanting to integrate the just-in-time and/or visualisation modules of the WATCHME project into their solutions, including - but not limited to - electronic portfolio systems such as EPASS.

2. Introduction

2.1 Glossary

Table 1 defines central terms and abbreviations used within this document.

Table 1: Glossary of terms

Term	Explanation
API	Application Programming Interface
EPASS	Electronic Portfolio and Assessment Support System
JIT	Just-in-Time feedback module
SM	Student Model
UI	User Interface
VIZ	Visualisation module

2.2 Background & scope of the deliverable

In accordance with requirements (cf. deliverable 3.1) and overall system architecture (cf. deliverable 3.2), this deliverable provides the detailed design of the interaction of both modules (JIT and VIZ) with the electronic portfolio system and the student-model module. Four separate interactions are specified:

- JIT/EPASS interaction: This involves the specification of how integration of JIT into the electronic portfolio user interface will be realised, and what data has to be exchanged between the e-portfolio system and the JIT module.

- VIZ/EPASS system interaction: This involves the specification of how integration of VIZ in the electronic portfolio user interface will be realised, and what data has to be exchanged between the e-portfolio system and the VIZ module.

- JIT/SM interaction: This involves the specification of applied communication protocol and data exchange during a request for feedback from the student model to the JIT module, either directly or through the e-portfolio system.

- VIZ/SM interaction: This involves the specification of applied communication protocol and data exchanged during the collection of data to be visualised.

As the requirements process is on–going beyond the date of this delivery, this deliverable describes the general integration mechanisms, but leaves open details specific to particular JIT/VIZ representations. These details will emerge through the development of the JIT/VIZ modules, and be documented as part of the report deliveries 5.2 (JIT) and 5.4 (VIZ).

2.2.1 Constraints

A number of goals and constraints not directly attributable to the initial WATCHME project work plan, but identified through dialogue between the technical partners, influence the architecture significantly:

- The JIT/VIZ modules must integrate naturally within the existing, web-based EPASS system.
- The JIT/VIZ modules should be embeddable within other portfolio systems and therefore not be tightly coupled to EPASS.
- The JIT/VIZ modules should support mobile devices, e.g. by gracefully degrading to simpler/less interactive visualisations.
- The JIT/VIZ modules should allow for additional types of visualisation or feedback to be implemented (e.g. to support specific needs of different application domains) post WATCHME.
- The architecture should allow EPASS and SM to evolve independently from the JIT/VIZ modules.

3. Common Architecture

Although the information provided by the JIT and VIZ modules differs, they share the trait that they need to be embedded into a hosting application, such as an electronic portfolio management system.

Consequently, the integration architecture for embedding said modules into a host application is shared between the JIT and VIZ modules. At the highest level, JIT and VIZ aspects are separated into three tiers, with external integrations taking place at the upper and lower tiers.

Figure 2 below gives a logical view of these tiers, with integrating artefacts (outside the scope of this deliverable) marked blue.

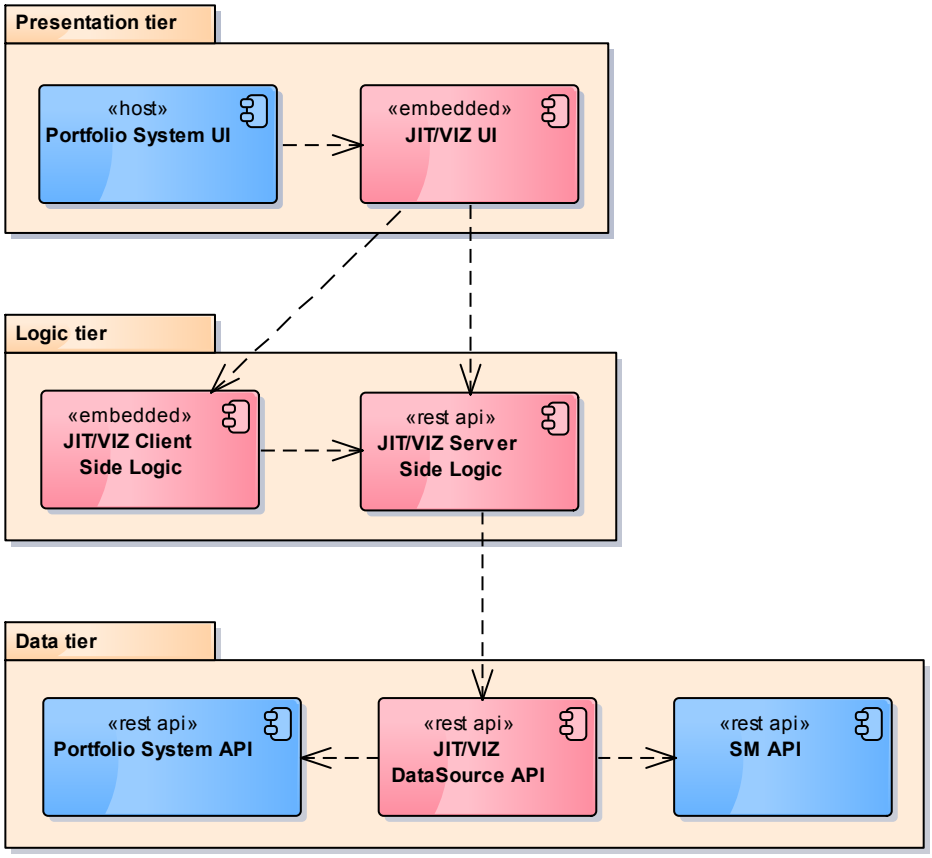

Figure 2: Logical tiered view of integration architecture

This separation into tiers is technology-agnostic, with the constraint that the JIT/VIZ UI and client side logic embedded into a host must be implemented using a technology compatible with the host. E.g. using JavaScript and HTML would be a natural choice for a web-based host. For a host on a mobile device, a browser-based host is also an option, or a native platform implementation technology such as Java or Objective-C could be ap-

plied. Alternatively, a hybrid approach based on an embedded web view in a native application is also an option.

Below, in Figure 3, this is illustrated by placing the artefacts of the three tiers according to an abstract deployment perspective grouped by independent technology stacks:

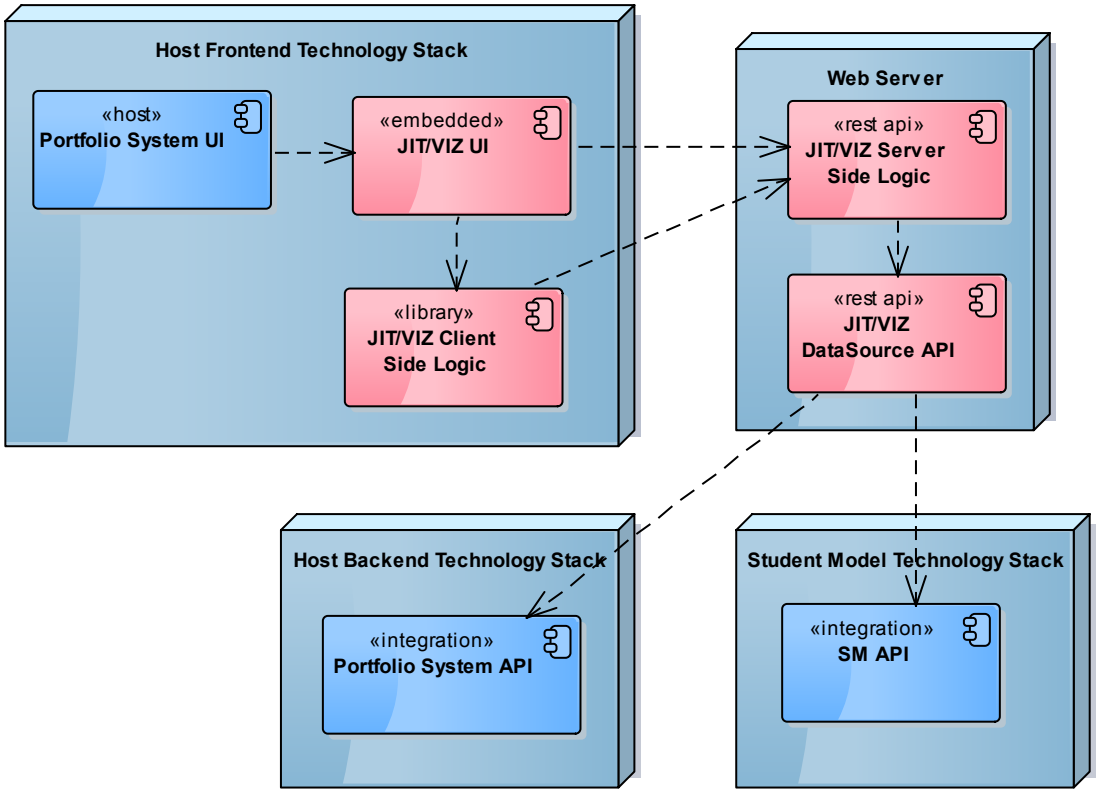

Figure 3: Logical deployment view of integration architecture

3.1 UI level integration

The approach to integrating both visualisations from VIZ and feedback from JIT into the portfolio system UI is to embed JIT/VIZ specific artefacts into the host UI.

From a structural perspective, this means:

- Referencing JIT/VIZ specific code from the host (e.g. JavaScripts in the page HTML if the host is web based, or e.g. Java JAR files if the host were based a mobile Android application)
- Embedding JIT/VIZ content placeholders in the host UI (e.g. HTML DIVs if the host is web based, or e.g. native Views/Fragments if the host were a mobile Android application)
- For interactive content, embedding JIT/VIZ affordances in the host UI (e.g. buttons and selectors) for obtaining user input to visualisations or feedback

Figure 4 provides the structural view of the different UI components and their integration. JIT/VIZ UI affordances are modelled as JIT/VIZ controls respectively, and content placeholders as JIT/VIZ canvases. Canvases can serve as controls themselves, to support visualisation/feedback scenarios where the user can interact directly with the content on the canvas, without external affordances. The JIT/VIZ controllers mediate between the UI affordances and the canvases. Each canvas has exactly one controller instance, but an instantiation-time choice can be made between several variants of controllers for a specific visualisation/feedback item, e.g. to gracefully degrade from an advanced version of a visualisation to a simpler version on legacy web browsers. An open number of controls can supply user input to the controllers, depending on the needs of a specific visualisation/feedback item.

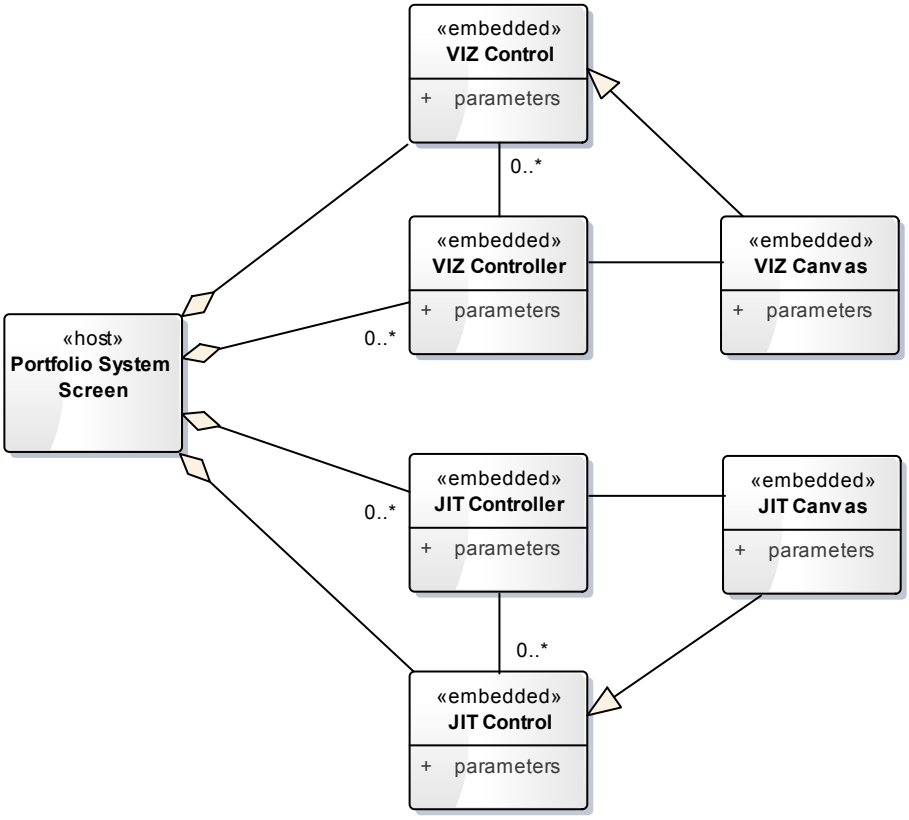

Figure 4: Structural view of UI integration architecture

Each artefact can be parameterised with contextual information such as user identification/session token, data sources and any instance specific configuration specific to a visualisation/feedback item.

The following table summarises the responsibilities of the artefacts in Figure 4:

Table 2: Responsibilities of main UI integration artefacts

Artefact	Responsibility
EPASS Page	Main user interface unit.
VIZ Controller	Visualisation specific mediator between canvas, controls and visualisation logic.
VIZ Canvas	Display of actual visualisation. Could be a container of DOM/SVG elements manipulated by the controller for advanced or interactive visualisations, or a simple bitmap container for simple visualisations.
VIZ Control	Optionally providing user adjustable input to the controller, allowing the visualisation to reflect that without doing a page reload.
JIT Controller	Feedback specific mediator between canvas, controls and just-in-time logic.
JIT Canvas	Display of feedback. Could be a container of CSS-styleable HTML markup for textual/in-portfolio system linking or of VIZ components if the feedback is a visualisation.
JIT Control	Optionally providing user adjustable input to the controller, allowing the feedback to reflect that without doing a page reload.

The integration has 2 modes, described in Table 3. Each mode implies a separate controller variant for the visualisation/feedback item in question, or a composite controller capable of handling both (implementation choice).

Table 3: UI integration modes

Mode	Nature	Description
Active	Client side	VIZ Controller obtains visualisations from client side implementations. Allows for interactive visualisations, but requires modern, unconstrained web browsers.
Passive	Server side	VIZ Controller obtains visualisations in image form, generated server side. Works on constrained/legacy browsers or report generation scenarios.

3.2 API level integration

The JIT and VIZ components are driven by data from the portfolio system and the student model. As these are separate entities (cf. deliverable 3.6 on the overall system architecture), the nature of the composed system is distributed. As it is a goal (cf. 2.2.1) to be able to apply JIT/VIZ modules independently of portfolio system and SM implementation, an architectural choice of isolating integration logic in a server side component is made. As a consequence, the JIT/VIZ client side logic can be simplified and focused on its primary responsibility, without dealing with details about how specific data for visuali-

sations and feedback items is provided by the concrete portfolio system and student model of a particular deployment.

The JIT/VIZ APIs uses a network-based application architecture, and is specifically based on the REST [1] architectural style. HTTP is the applied application protocol, transported over TCP/IP, with at least JSON [2] being supported for resource representations.

This is illustrated in the following component diagram:

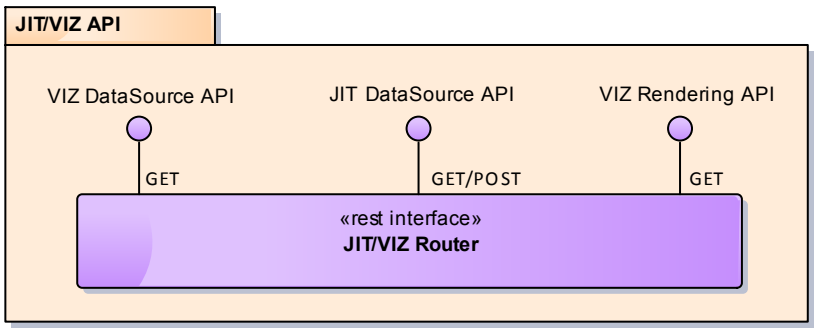

Figure 5: JIT/VIZ API endpoints

Table 4: API integration interfaces

Interface	Nature	Responsibility
VIZ DataSource	Read	Provides data series for consumption by concrete visualisations. Data sources do not directly reflect the underlying data sources (e.g. portfolio system repository) but expose data series relevant to a particular class of visualisation. Data sources must be configured for a particular deployment of the system.
JIT DataSource	Read/Write	Provides action sets (e.g. "Take assessment") for consumption by concrete feedback implementations. Data sources (e.g. "What should I do next?") must be configured for a particular deployment of the system. Optionally supports receiving notification from a JIT controller about a particular action taken by the user, in the event that the SM integration needs to be notified outside the SM-portfolio system integration.
VIZ Rendering	Read	Provides optional off-client rendering support for visualisations that cannot be handled natively for a particular client type (cf. passive mode in Table 3).

3.2.1 DataSource APIs

The responsibility of the JIT/VIZ DataSource API is to provide a uniform data source abstraction for, and at the same abstraction level as, the JIT and VIZ modules, mapping those to concrete repositories of data in a given deployment, such as EPASS and SM.

The actual layout of the requests and responses is not specified in this deliverable, although it is specified that the information will have JSON representations.

VIZ DataSource

The API should be able to supply the following kinds of information.

- Information about a specific student (e.g. assessments by self and others)
- Group-level information (e.g. students in a specific class or year, or for a particular subject)
- Aggregated information (e.g. average scores of students for a given EPA and time period)

JIT DataSource

The JIT API enables an interactive environment through which a list of currently relevant feedback messages for the user, based on her student model, can be presented. Feedback messages can be actionable and acting on suggestions can update the state of the student model, either indirectly through triggering an action in the portfolio system that leads to new/updated data being transmitted to the student model through its portfolio system integration, or directly through notifying the student model API that a given action was taken.

From a behavioural perspective, the general flow of this integration is illustrated in the sequence diagram of Figure 6, below:

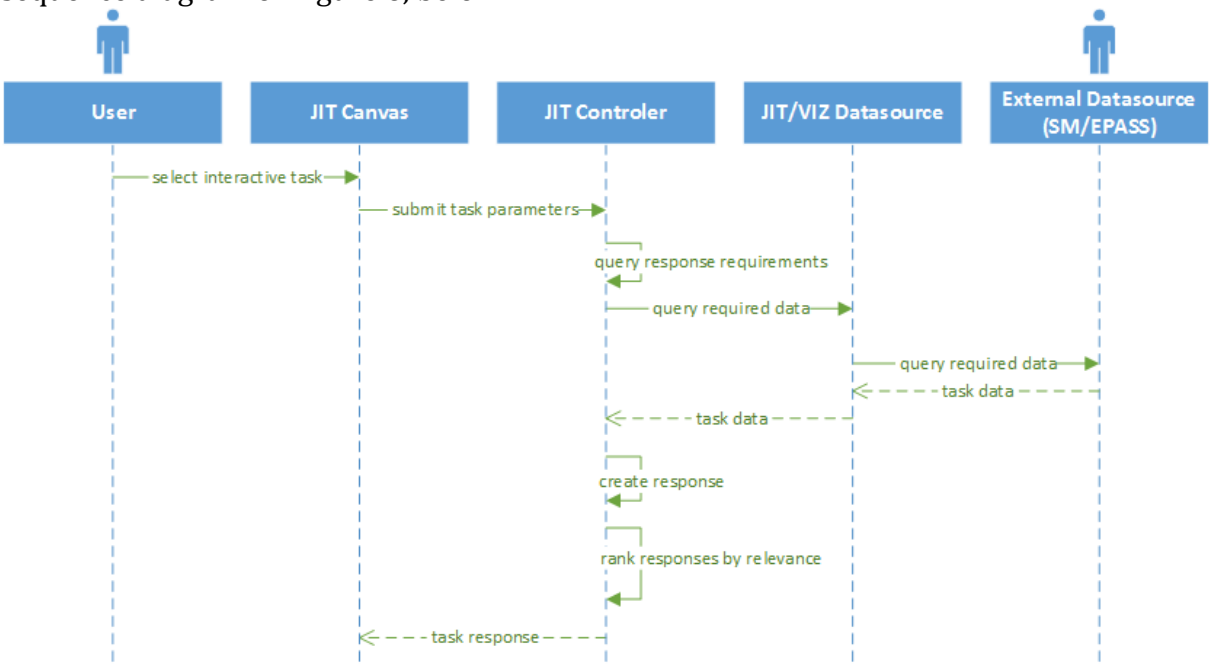

Figure 6: Behavioural view of feedback interaction

In the optional case of the JIT controller needing to directly notify the SM of the action taken (cf. JIT DataSource in

Table 4 above), the interaction would be extended by the act of the user selecting a task response on the JIT Canvas triggering a notification to the JIT Controller, which will send a message to the SM via the data source.

The following scenarios exemplify the kinds of interactions provided by this API:

Scenario 1: On what competency should I focus next?
For this scenario, there are several possible actions:

a. Display the list of competencies, ranked by the lack of performance in a certain discipline. The following steps are required to display this action:
   i. Query the SM for all the available **competencies**;
   ii. □ c □ **competencies**, query the SM for the **performance** indicator of **c**
   iii. Rank and select top n competencies by the lack of performance
   iv. Compile and display the list. **Selecting an item from this list will trigger a portfolio system action (e.g. review the selected competency).**
b. Display the graph of performance, per competency. **This selection will trigger a VIZ action.**
c. Review your portfolio today. **This selection will trigger a portfolio system action.**
d. Do something else. **This selection will trigger a JIT action.**

Scenario 2: What should I do next?
For this scenario, there are several possible actions:

a. Review your portfolio today. **This selection will trigger a portfolio system action.**
b. Review the positive feedback received from your supervisor. (Active if positive feedback has been received in the last **n** days). *Query SM for positive feedback.* **This selection will trigger a portfolio user action.**
c. Review the negative feedback received from your supervisor. (Active if negative feedback has been received in the last **n** days). *Query SM for negative feedback.* **This selection will trigger a portfolio user action.**
d. Update or create a self-assessment form. (Active if no self-assessment has been received in the last **n** days). **This selection will trigger a portfolio user action.**
e. Display the global overview graph. **This selection will trigger a VIZ action.**
f. Display the list of competencies, ranked by the lack of performance in a certain discipline. **This selection will trigger a portfolio user action.**
g. Display the graph of performance, per competency. **This selection will trigger a VIZ action.**
h. Review your portfolio today. **This selection will trigger a portfolio user action.**
i. Do something else. **This selection will trigger a JIT action.**
j. Etc.

3.2.2 Rendering API

The responsibility of the VIZ Rendering API is to render server side visualisations (i.e. the passive mode of integration described in Table 3 of section 3.1). It is stateless and able to respond with a static image, given relevant parameters. That image can then be used as the source of e.g. an HTML `<img>` element in the client web page, or consumed by native image components on e.g. mobile devices. As a result, there will be no further user interaction. This is required to support clients with little or no JavaScript or SVG capabilities, such as older mobile browsers or historical versions of Internet Explorer.

As a minimum, the Portable Network Graphics (PNG) image format must be supported (MIME type `image/png`) [3].

3.3 Security & Privacy

Authentication and authorisation are already concerns on the portfolio system. With JIT and VIZ being embedded into the portfolio system's user interface, it is natural to propagate authentication and authorisation concerns for the JIT/VIZ modules to the portfolio system as well.

The JIT/VIZ APIs do not communicate actively with the portfolio system to achieve this, but delegate the responsibilities to the portfolio system API and SM API with which they integrate.

After authenticating a user in the hosting portfolio system, it will pass an opaque session context to the components embedded within it. This context, validatable by the portfolio system, will propagate, as a token passed to the JIT/VIZ controllers, to all requests made against the JIT/VIZ API interfaces. From there, propagation continues to the underlying portfolio system and SM APIs (in the latter case, propagation continues to the portfolio system), where the token is ultimately verified through the portfolio system's Privacy Manager to ensure that requests being made against them are allowed in the given context (authorisation).

Since the token might expire between the time it is issued by the portfolio system and its first use, the JIT/VIZ API should recognise this, and return a "token expiry" result back to the JIT/VIZ controller, which can then signal the user to refresh the page (or take other measures, if available, for obtaining a fresh token). Similarly, controllers must be prepared to handle authorisation errors, and communicate these to the user.

Information privacy is handled at the transport level (i.e. by TLS/SSL).

4. JIT/VIZ API description

There are two major categories of components in JIT/VIZ ecosystem: client-side UI components and server-side REST API resources. The latter category constitutes the JIT/VIZ API and will be consumed solely by the UI controllers of the former category.

This section maps the logical concepts of section 3.2 to concrete REST resources.

Figure 5 (section 3.2) illustrated 3 logical surfaces of the JIT/VIZ API: JIT DataSource API, VIZ DataSource API and VIZ Rendering API. The two DataSource APIs serve the same purpose of acting as glue between a module and data from an underlying API. As they are logically without intersection (JIT and VIZ modules can exist independently), they are mapped to two distinct resource paths in the JIT/VIZ API. Authorisation is handled by these underlying systems; for this an authentication token is propagated.

The VIZ Rendering API can be regarded as merely a different resource representation of a visualisation, and is therefore mapped to the same resource path as the VIZ DataSource API.

4.1 JIT resources

The JIT resources of the JIT/VIZ REST API interface are consumed solely by client-side JIT widgets. Upon request, the SM API is interrogated and a JSON serialisation of feedback and actions for the current student is provided.

The available resources are described next.

`/API/JIT/Questions?AuthToken={AuthToken}&{Parameters}`

Used to obtain all questions available in JIT. UI will need these to offer a selection to the end users.

`GET` Method:

- Resource Parameters:
  - Questions: the just-in-time feedback questions.
- Query Parameters:
  - *AuthToken*: represents the authentication token passed on for each request,
  - *Parameters*: any relevant parameters for JIT process (Optional).
- Available responses:
  - 200 (OK) – JSON; List with all possible questions.
  - Error: through standard 4xx (e.g. 400 Bad request, 404 Not found, etc.) and 5xx (e.g. 500 Internal Server Error etc.) HTTP status codes [4] which will indicate that either the client input was incorrect or that the server computations faulted.

`/API/JIT/Feedback/QuestionId?AuthToken={AuthToken}&{Parameters}`

Used to query update the JIT system for actions to a specific question chosen by the user. Also optionally allows for notifying JIT directly that a particular action was taken.

`GET` Method:

- Resource Parameters:
  - Feedback/QuestionId: resource that represent the list of actions to a specific question identified through *QuestionId*.
- Query Parameters:
  - *AuthToken*: represents the authentication token passed on for each request
  - *Parameters*: Identification of action taken
- Available responses:
  - 200 (OK) – JSON; Gathered response from SM (in form of actions to be taken).
  - Error: through standard 4xx (e.g. 400 Bad request, 404 Not found, etc.) and 5xx (e.g. 500 Internal Server Error etc.) HTTP status codes [4] which will indicate that either the client input was incorrect or that the server computations faulted.

`POST` Method:

- Resource Parameters:
  - Feedback/QuestionId: resource that represent the list of actions to a specific question identified through *QuestionId*.
- Query Parameters:
  - *AuthToken*: represents the authentication token passed on for each request
  - *Parameters*: Additional parameters that might be needed for each particular question (Optional).
- Available responses:
  - 200 (OK)
  - Error: through standard 4xx (e.g. 400 Bad request, 404 Not found, etc.) and 5xx (e.g. 500 Internal Server Error etc.) HTTP status codes [4] which will indicate that either the client input was incorrect or that the server computations faulted.

4.2 VIZ resources

The Visualisation API will be a REST [1] API interface consumed solely by client-side visualisation widgets. Upon request, the EPASS (and potentially SM) API is interrogated and a JSON serialisation of the information necessary for the visual widget (static or interactive version) to render properly.

Visualisation API will support only a single REST verb: GET

`/API/Visualizations/{VisualizationType}?AuthToken={AuthToken}&{Parameters}`

`GET` Method:

- Resource Parameters:
  - *VisualizationType*: represents the widget type that will be supported by the VIZ implementation

- Query Parameters:
  - *AuthToken*: represents the authentication token passed on for each request,
  - *Parameters*: the configuration parameters that might be needed by each particular visualisation (may consist in visual information like: width, height, xAxisLabel, zAxisLabel etc. or any other settings).
- Available responses:
  - 200 (OK) – JSON; Necessary information for the widget to be properly rendered.
  - Error: through standard 4xx (e.g. 400 Bad request, 404 Not found, etc.) and 5xx (e.g. 500 Internal Server Error etc.) HTTP status codes [4] which will indicate that either the client input was incorrect or that the server computations faulted.

For each visualisation request, the Visualisation API will start a workflow with the following actions:

1. Access EPASS API and retrieve the appropriate data, passing on the authorisation token (plus any other contextual information), in order to get the necessary portfolio data.

2. Aggregate the received data according to the visualisation type that was requested.
   In this phase the received data should be adapted to simple series of tuples ready to be used in the visualisation process.

3. Build the visualisation response.
   Visualisation widget rendering can either be done on client-side or on server-side.
   For client-side rendering the VIZ API should provide only the necessary series data and the client-side logic should handle the rest.
   Server-side rendering comes with a performance gain for low-end clients or clients without advanced rendering capabilities at the cost of rich and interactive graphical display.

Taking into account the different ways in which the widgets might be rendered on the client side, the API response may contain one of the following:

- Serialised JSON data only, that will be used directly on client side,
- HTML/JS code built on the server side that may be used directly on the client,
- Static image which may represent directly the requested visualisation.

HTTP content negotiation (specifically `Accept/Content-Type` headers) is used to designate the desired type of representation.

5. EPASS integration

Figure 7 shows a concrete application of the tiering described in Figure 2 of section 3, in the context of the web-based portfolio management system EPASS, accessed through a desktop or mobile web browser.

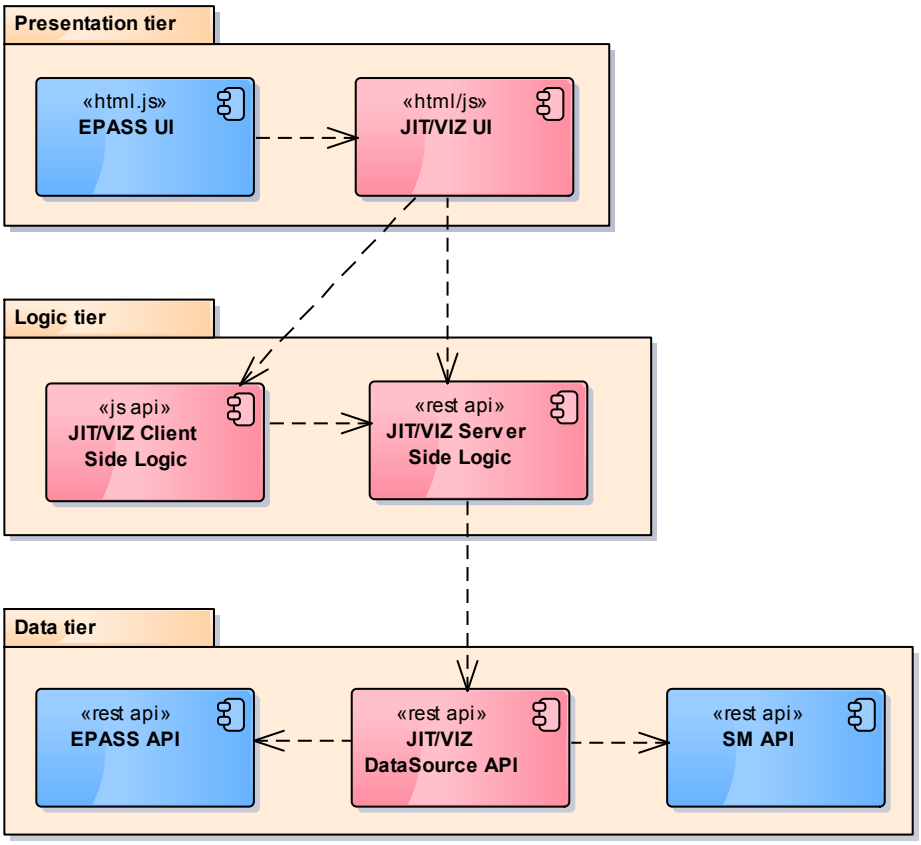

Figure 7: Logical view of integration architecture as applied to EPASS

Similarly, Figure 8 below shows a concrete instance of the logic deployment perspective from Figure 3, in the context of EPASS as hosting portfolio system.

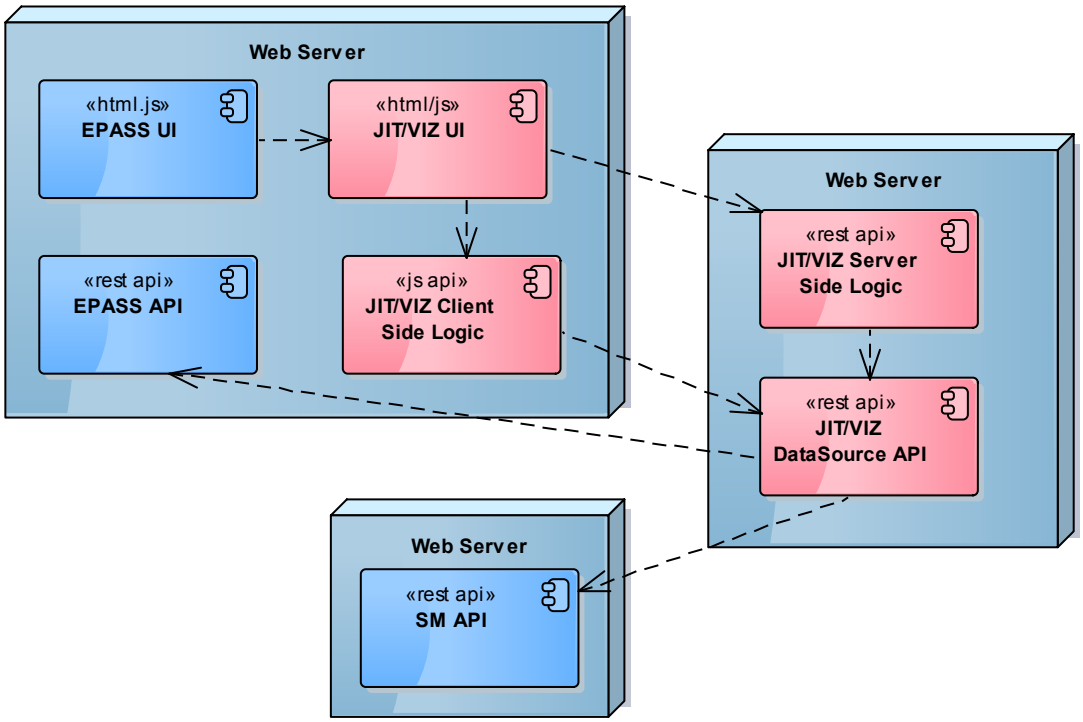

Figure 8: Deployment view of integration architecture as applied to EPASS

An actual deployment may choose to separate or consolidate the web server instances onto one or several physical instances, e.g. to address scalability or to decouple deployment dependencies. Cross Origin Resource Sharing (CORS) [5] can be applied to enable cross-domain communication from the browser, in deployments where EPASS and APIs do not reside on the same internet domain.

5.1 GUI integration

To include the JIT/VIZ modules in EPASS, EPASS will add DIV elements to the pages when it's generated at server side. These DIV elements will act as placeholders for code generated by the JIT/VIZ modules. The HTML inside the DIV elements will be generated through JavaScript code included from the JIT/VIZ modules. EPASS will also include a CSS file containing the core styling needed for the JIT/VIZ module. The EPASS API will produce links for portfolio actions as described in scenario 2 (section 3.2.1).

EPASS will include one or more controller JavaScript files from the JIT/VIZ module, which should provide functions to display the visualisation or feedback. These JavaScript functions should be able to accept parameters (e.g. user hash, the type of visualisation, the question to be answered by the feedback, etc.) and communicate with the JIT/VIZ REST API using CORS.

Figure 9 illustrates the embedding of JIT/VIZ controls into an EPASS webpage:

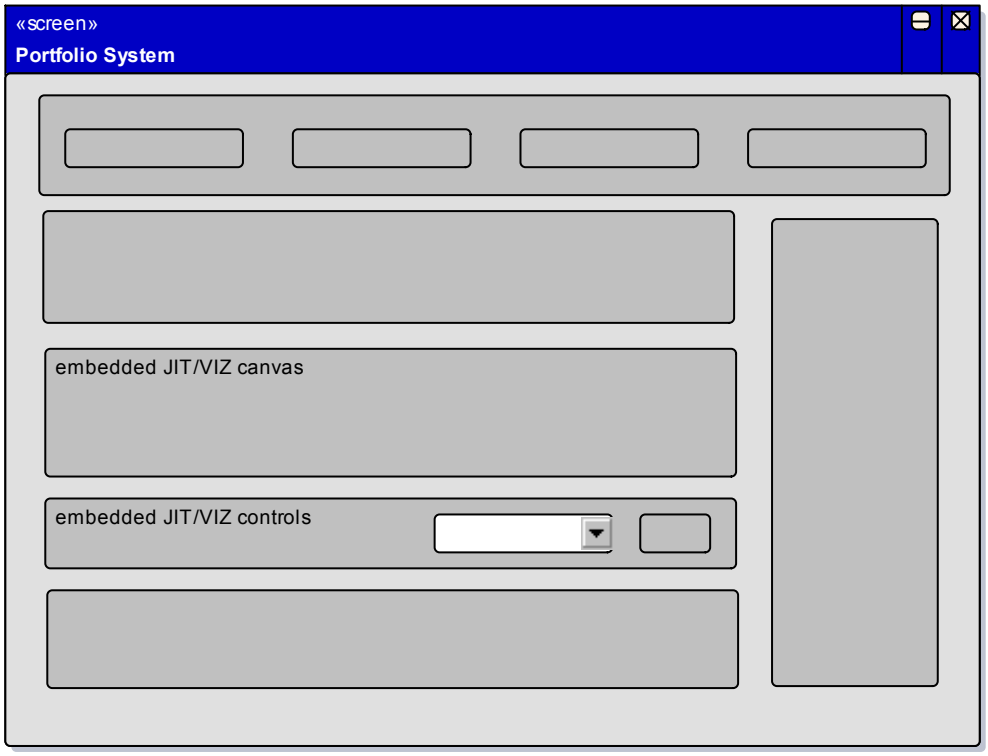

Figure 9: Embedded JIT/VIZ controls/canvas in an EPASS webpage

As applied to EPASS, the structural view of this integration (the general form of which was shown in Figure 4) appears in Figure 10 below.

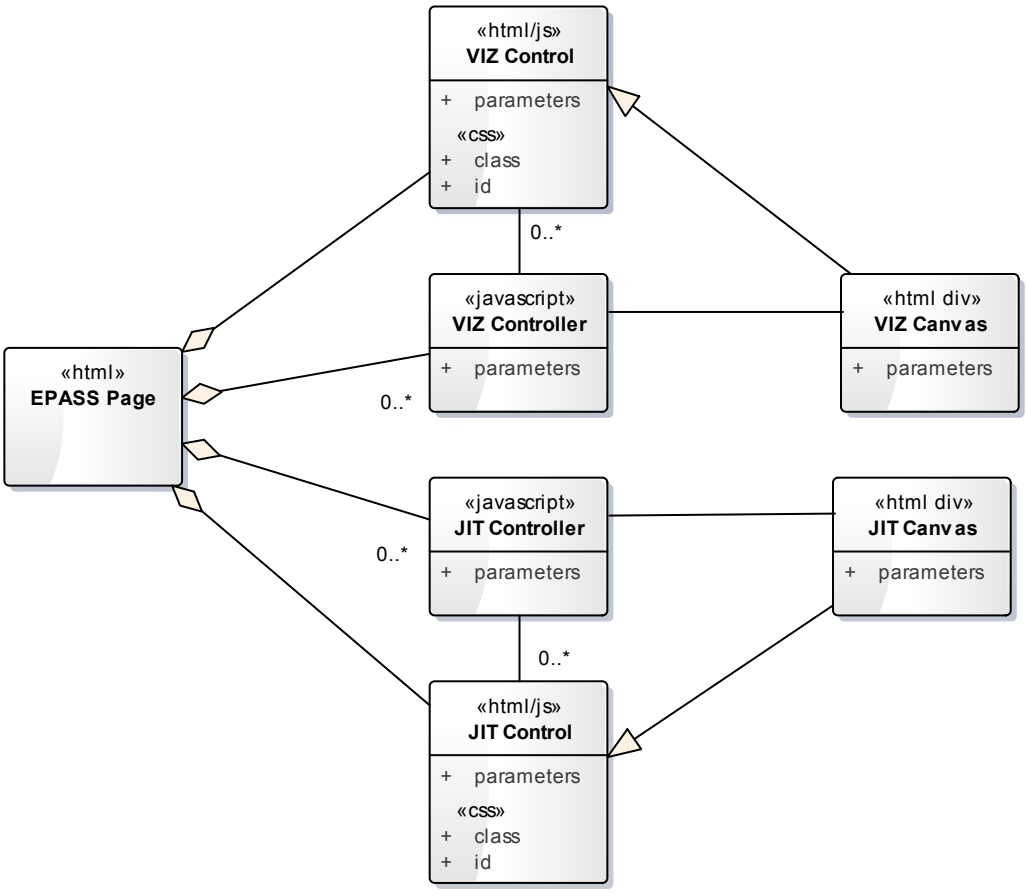

Figure 10: Structural view of UI integration architecture as applied to EPASS

From a behavioural perspective, the sequence diagram of Figure 11 illustrates how these artefacts are tied together during a page load in EPASS:

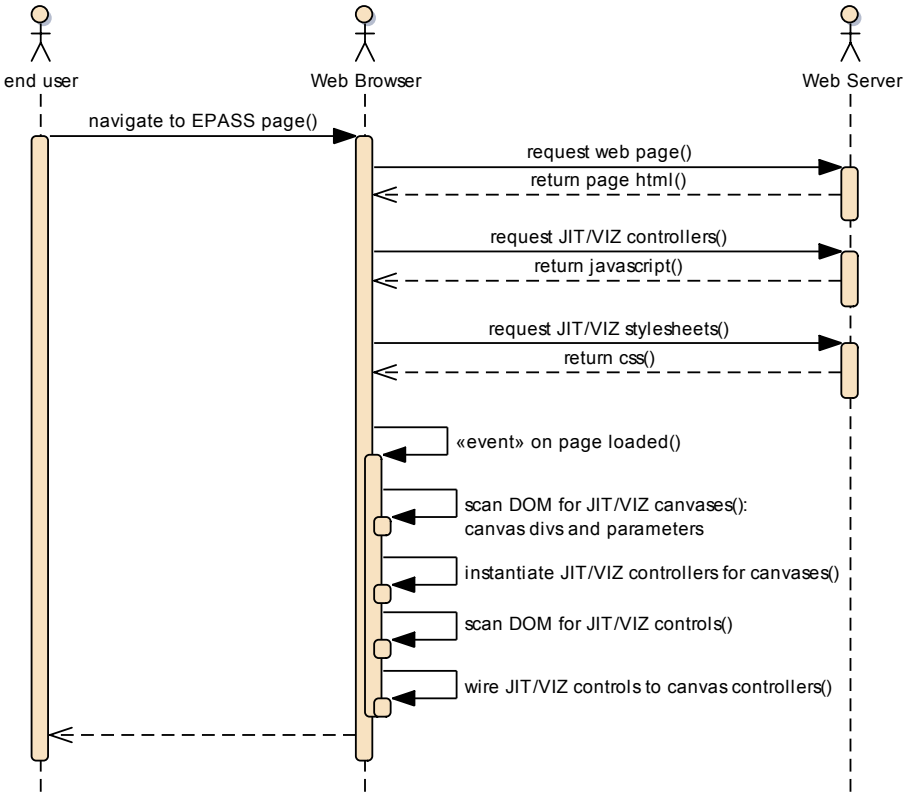

Figure 11: Behavioural view of EPASS UI integration

An example HTML skeleton page embedding a visualisation is provided in the following code snippet:

```
<html>
<head>
…
<script type="text/javascript" src="watchme_viz.js" />
<link rel="stylesheet" type="text/css" href="watchme_viz.css" />
…
</head>
<body>
…
<div id="student_cohort_spiderchart"
      class="viz_spiderchart"
      data-source="API-DATASOURCE"
      data-mode="filled"
/>
…
<script type="text/javascript">
      watchme.viz.init() // Instantiate controllers & wire controls
</script>
…
</body>
</html>
```

5.2 API integration

EPASS will provide a REST-based API exposing relevant (e.g. competencies and assessment time series), and potentially aggregated (e.g. cohort averages) data to the JIT/VIZ API. As minimum, resources can be represented in JSON format.

The API will accept and verify a client token parameter to ensure that a request is allowed in the current context.

For details about the EPASS API, the reader is referred to deliverable 3.2 on the general architecture.

6. Student Model integration

This section provides a high-level outline of the SM perspective on the JIT/VIZ API integration. Refer to deliverable 4.1 for a more detailed description.

According to deliverable 4.1, the preliminary concepts regarding the Student Model API include several components:

- **External Student Model API** – which is used to link the SM to the EPASS and JIT/VIZ API. This API is described in the next sections of this deliverable.
- **Data (pre- and post-) Processing Module** – used to communicate with the student model external API and convert the data from/in internal student model structures. This component will be described in more detail in deliverable 4.2.
- **Bayesian Student Model** – comprises the domain and individual student models. The structure of this component will be described in more detail in deliverable 4.2.
- **Student Model Database –** could potentially store the student models and different parameters required at runtime. The component will be described in more detail in deliverable 4.2.

Figure 12 presents the interaction between the Student Model API and different internal and external modules.

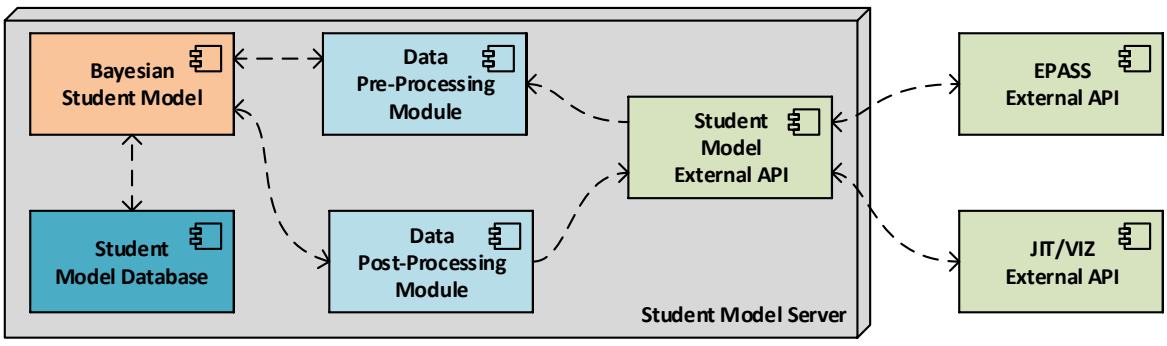

Figure 12: Student Model API component diagram

Preliminary notes

- The Student Model External API will use a RESTful API
- EPASS can push and query data from the SM: HTTP GET, POST and DELETE
- VIZ module can query, filter or aggregate student model output: HTTP GET
- JIT module can query, filter, aggregate or post data: HTTP GET and POST
- JIT/VIZ modules use a common layer to communicate with the Student Model

API description

The SM would expose 2 methods corresponding to different operations:

1. Query data, which can be used to retrieve the latest image of a given data type for a certain student or cohort of students. The query API specifies the student and/or group, domain and the required data.
2. Post data, support several operations (i.e. Update, Replace, Rollback or Delete). These operations correspond to different functions supported by the SM.

Authorisation and Privacy Management will be kept by EPASS, by using existing APIs. More details regarding the SM API are provided in the architecture design document of deliverable 3.2.

7. Conclusion

Deliverable 5.1 has served as catalyst for achieving alignment between the technical partners of the WATCHME project with regards to the architecture and overall design of the integration between the JIT and VIZ modules, the SM module and a hosting portfolio system such as EPASS, within the constraints imposed by these respective subsystems.

The present document has captured the central decisions and models, while leaving open enough details to accommodate the individual needs of requirements as they are being progressively elicited, and will form the foundation upon which the actual JIT and VIZ implementations will later be realised.

Further details

The architecture of the JIT and VIZ modules fit within the larger context of the general WATCHME system architecture. This is described in more details in deliverable 3.2

The JIT/VIZ API integrations rely on the EPASS and SM APIs. The external interface of both are described in more detail in deliverable 3.2.

Implementation details of the actual data fields and JSON representation of the JIT/VIZ data sources are described in deliverables 52 (JIT) and 5.4 (VIZ).

8. References

[1] R. T. Fielding, "Representational State Transfer (REST)," 2000. [Online]. Available: http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

[2] Ecma International, "The JSON Data Interchange Format (ECMA-404)," October 2013. [Online]. Available: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf.

[3] W3C, "PNG (Portable Network Graphics)," [Online]. Available: http://www.w3.org/Graphics/PNG/.

[4] IETF, "HTTP 1.1 RFC2616 Client Error 4xx, Server Error 5xx," [Online]. Available: https://tools.ietf.org/html/rfc2616#section-10.4, https://tools.ietf.org/html/rfc2616#section-10.5.

[5] W3C, "Cross-Origin Resource Sharing," 2014. [Online]. Available: http://www.w3.org/TR/cors/.

9. Tables and Figures

9.1 List of tables

9.2 List of figures

10. History of the document

10.1 Document history

Version	Author(s)	Date	Changes
0.1	Mads Troest	2014-09-26	Initial structure and content
0.2	Mads Troest	2014-10-10	Feedback from draft 0.1 incorporated
0.3	Mads Troest	2014-10-17	Feedback from draft 0.2 incorporated
0.4	Mads Troest Steen Lehmann	2014-10-31	Elaborated S3, extracted some content to S5 Elaborated S3.3
0.5	Mads Troest Ovidiu □erban Netrom Rik Wijnen	2014-11-14	Feedback from draft 0.4 incorporated Contributed JIT scenarios to to S3.2.3 Contributed S4 Contribution to S5.1
0.6	Ovidiu Serban, Jaime Costa, Atta Badii, Dan-iel Thiemert	2014-11-18	Contributed SM API overview to S6
0.7	Netrom	2014-11-21	Updated API descriptions in S4.1.2 and S4.1.3
	Mads Troest	2014-11-24	Updated S3.3
0.8	Mads Troest Ovidiu Serban	2014-12-08	S2.3 (Scope) merged with S2.2 (Background) S3 restructured S4 restructured, terminology aligned with S3 Updated figures 2, 3 and 7 (Netrom feedback) Fixed references Contributed sequence diagram to S3.2.3 Updated SM API overview in S6
0.9	Mads Troest	2014-12-16	Incorporated feedback from UoR review
1.0	Mads Troest	2014-12-16	Incorporated feedback from UM review

10.2 Internal review history

Internal Reviewer	Date	Comments
Daniel Thiemert (UoR)	2014-12-09	Minor corrections and clarifications needed, proof-read the document
Jeroen Donkers (UM)	2014-12-16	Corrections on deliverable numbering Suggestion on EPASS integration
Martijn Holthuijsen (Mateum) Rik Wijnen (Mateum)	2014-12-16	Reviewed document