IRP Adapter

Developer’s Guide

Prepared by UCLA/CRESST

Ernesto De La Luz Martinez

Table of Contents

1.1 Design Class Diagrams.

The Design class diagram for the TDS Reports RESTful service is described below; this diagram contains all the classes involved in the process of automation reports.

Class AdapterController:

This class is a controller that handles the HTPP and is annotated with the @RestController annotation.

Interface AdapterAutomationService.

This interface is contains the next methods:

1) automate(). This method starts the automated generation of TDSReports.
2) getAdapterAutomationTicket(UUID adapterAutomationToken). Gets the latest update of the AdapterAutomationTicket for the given Automation Token.
3) getTdsReports(Date startTimeOfSimulation). Gets all of the generated Ids of the TDSReports starting from the given date and time or since the time of previous test Simulation.
4) getTdsReport(int tdsReportId). Gets an individual TDSReport, according to the tdsReportId passed as parameter.

Class SbossAutomationAdapterService.

This class is the implementation of the AdapterAutomationService interface.

Class AdapterAutomationTicket.

This class represents a ticket used for checking the status of the automation process. Also is embedded within this class a property of type AdapterAutomationStatusReport, which is snapshot of the current progress made by the automation engine.

Class AdapterAutomationStatusReport.

This class contains the data of the current status of the TDS report, made by the automation engine.

Class AutomationEngine.

This class handles IRP Automation request, making sure only a single execution of automation against a vendor system is run. Also it runs IRP Automation asynchronously while handling messages back to the client, using the AdapterAutomationTicket to populate the status of the automation process.

Class SbossAutomationEngine.

This class implements the interface AutomationEngine.

Class TdsReportExtractor.

This class has the methods to extract TDS Reports from the Test Delivery System.

Class DocumentXmlRepository.

This class is the repository for the operations related to the table "XMLRepository" and the view "v_MostRecentXml".

Class DocumentXmlRepositoryImpl.

This class implements the interface DocumentXmlRepository.

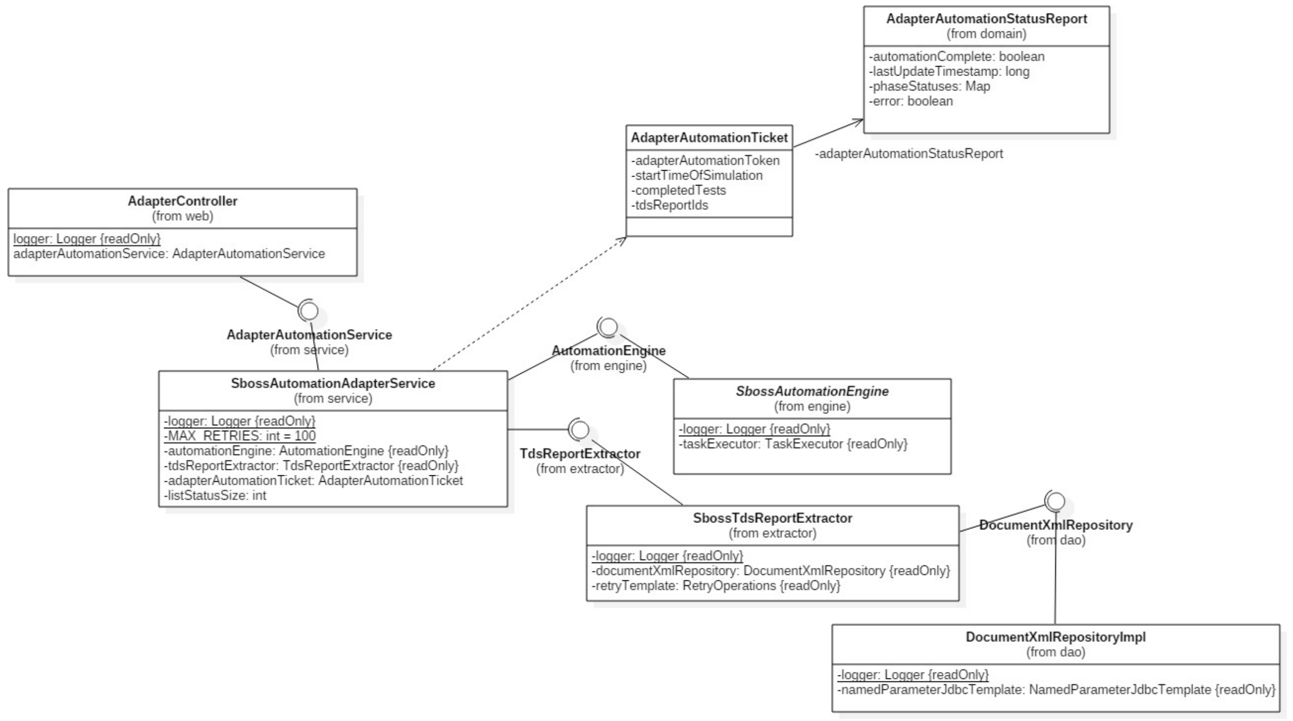
AdapterAutomationStatusReport
(from domain)
-automationComplete: boolean
-lastUpdateTimestamp: long
-phaseStatuses: Map
-error: boolean
AdapterAutomationTicket
-adapterAutomationToken
-startTimeOfSimulation
-completedTests
-tdsReportIds
-adapterAutomationStatusReport
AdapterController
(from web)
logger: Logger {readOnly}
adapterAutomationService: AdapterAutomationService
AdapterAutomationService
(from service)
AutomationEngine
(from engine)
SbossAutomationAdapterService
(from service)
-logger: Logger {readOnly}
-MAX_RETRIES: int = 100
-automationEngine: AutomationEngine {readOnly}
-tdsReportExtractor: TdsReportExtractor {readOnly}
-adapterAutomationTicket: AdapterAutomationTicket
-listStatusSize: int
SbossAutomationEngine
(from engine)
-logger: Logger {readOnly}
-taskExecutor: TaskExecutor {readOnly}
TdsReportExtractor
(from extractor)
SbossTdsReportExtractor
(from extractor)
-logger: Logger {readOnly}
-documentXmlRepository: DocumentXmlRepository {readOnly}
-retryTemplate: RetryOperations {readOnly}
DocumentXmlRepository
(from dao)
DocumentXmlRepositoryImpl
(from dao)
-logger: Logger {readOnly}
-namedParameterJdbcTemplate: NamedParameterJdbcTemplate {readOnly}

Figure 7. Design class diagram of the TDS Reports RESTful service.

1.2 Sequence Diagrams

1.2.1 Automation Process(Client Side)

The next diagram describes the process used for the automation in the client side.

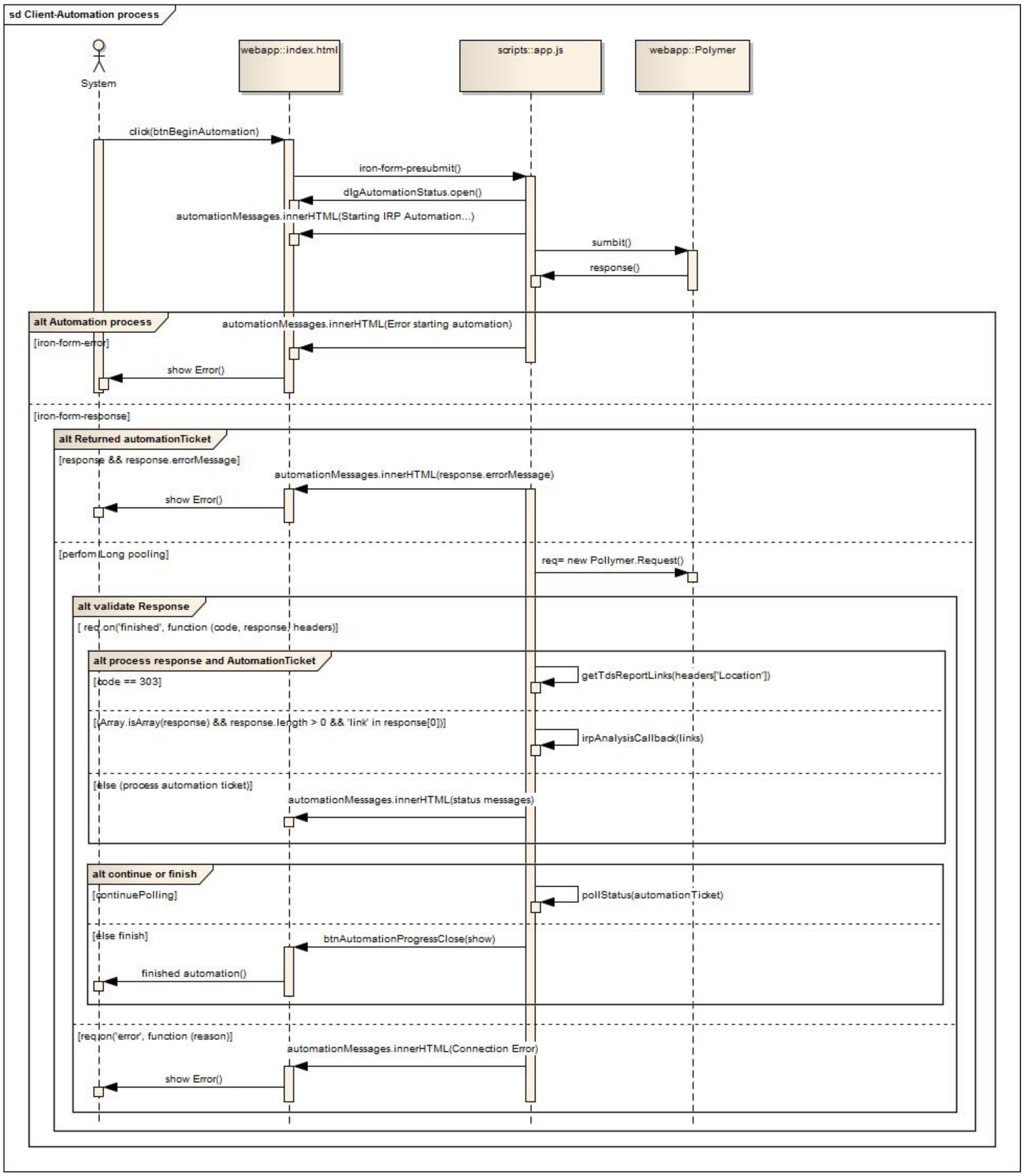
sd Client-Automation process
System
webapp::index.html
scripts::app.js
webapp::Polymer
click(btnBeginAutomation)
iron-form-presubmit()
dlgAutomationStatus.open()
automationMessages.innerHTML(Starting IRP Automation...)
sumbit()
response()
alt Automation process
[iron-form-error]
automationMessages.innerHTML(Error starting automation)
show Error()
[iron-form-response]
alt Returned automationTicket
[response && response.errorMessage]
automationMessages.innerHTML(response.errorMessage)
show Error()
[perfom Long pooling]
req= new Pollymer.Request()
alt validate Response
[req.on('finished', function (code, response, headers)]
alt process response and AutomationTicket
[code == 303]
getTdsReportLinks(headers['Location'])
[(Array.isArray(response) && response.length > 0 && 'link' in response[0])]
irpAnalysisCallback(links)
[else (process automation ticket)]
automationMessages.innerHTML(status messages)
alt continue or finish
[continuePolling]
pollStatus(automationTicket)
[else finish]
btnAutomationProgressClose(show)
finished automation()
[req.on('error', function (reason)]
automationMessages.innerHTML(Connection Error)
show Error()

Figure 9. Diagram of Automation process in the client side.

1.2.1.1 Initiate the Automation process

Details of each process in the diagram are explained below:

The automation process is triggered by the client:

1) The user click on the button "btnBeginAutomation".
2) The "iron-form-presubmit" even listener is triggered.
    a. The dialog for displaying the automation status process is opened.
    b. A message indicating that the automation process is started is added to the screen.
3) The form is submitted
4) An object response is returned from the server side.

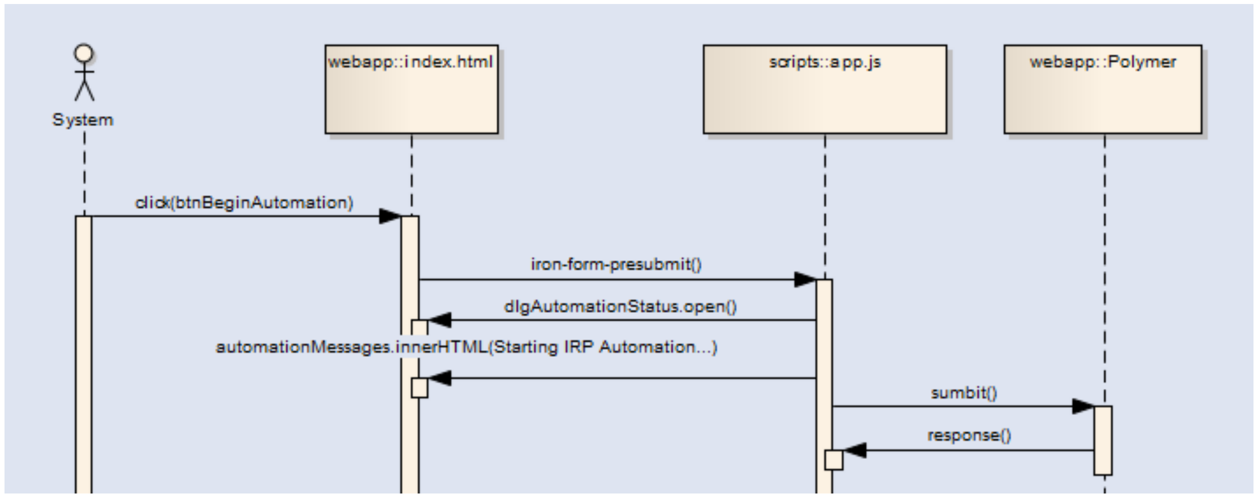
System
webapp::index.html
scripts::app.js
webapp::Polymer
click(btnBeginAutomation)
iron-form-presubmit()
dlgAutomationStatus.open()
automationMessages.innerHTML(Starting IRP Automation...)
sumbit()
response()

Figure 10. Diagram when the automation process starts.

1.2.1.2 Automation Process

In this step the response is processed in two different statuses:

1) Iron-form-error. If this event is triggered it means there was an error on the response; which means the automation process is ending, displaying an error message to the user.
2) Iron-form-response. When this event is triggered, the automation process has started.

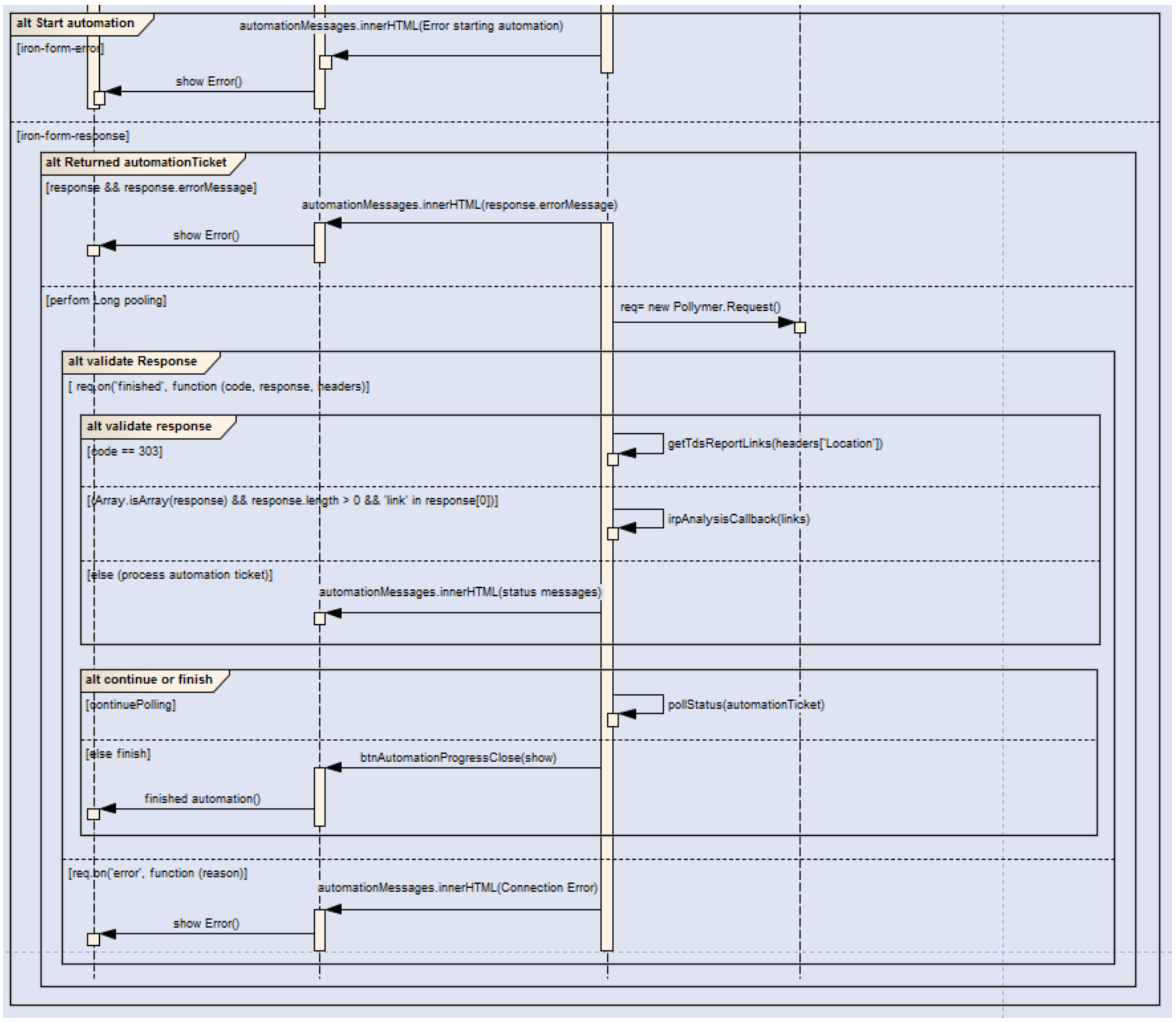
alt Start automation
[iron-form-error]
automationMessages.innerHTML(Error starting automation)
show Error()
[iron-form-response]
alt Returned automationTicket
[response && response.errorMessage]
automationMessages.innerHTML(response.errorMessage)
show Error()
[perfom Long pooling]
req= new Pollymer.Request()
alt validate Response
[req.on('finished', function (code, response, headers)]
alt validate response
[code == 303]
getTdsReportLinks(headers['Location'])
[(Array.isArray(response) && response.length > 0 && 'link' in response[0])]
irpAnalysisCallback(links)
[else (process automation ticket)]
automationMessages.innerHTML(status messages)
alt continue or finish
[continuePolling]
pollStatus(automationTicket)
[else finish]
btnAutomationProgressClose(show)
finished automation()
[req.on('error', function (reason)]
automationMessages.innerHTML(Connection Error)
show Error()

Figure 11. Diagram of the automation Process.

1.2.1.3 Validating the response

Once the response is returned, a validation is performed over this object:

1) If the response contains a property named "errorMessage", then an error is displayed to the user and the process is finished.
2) If the process is correct then a Request object is created to manage the long pooling, and additional steps are performed.

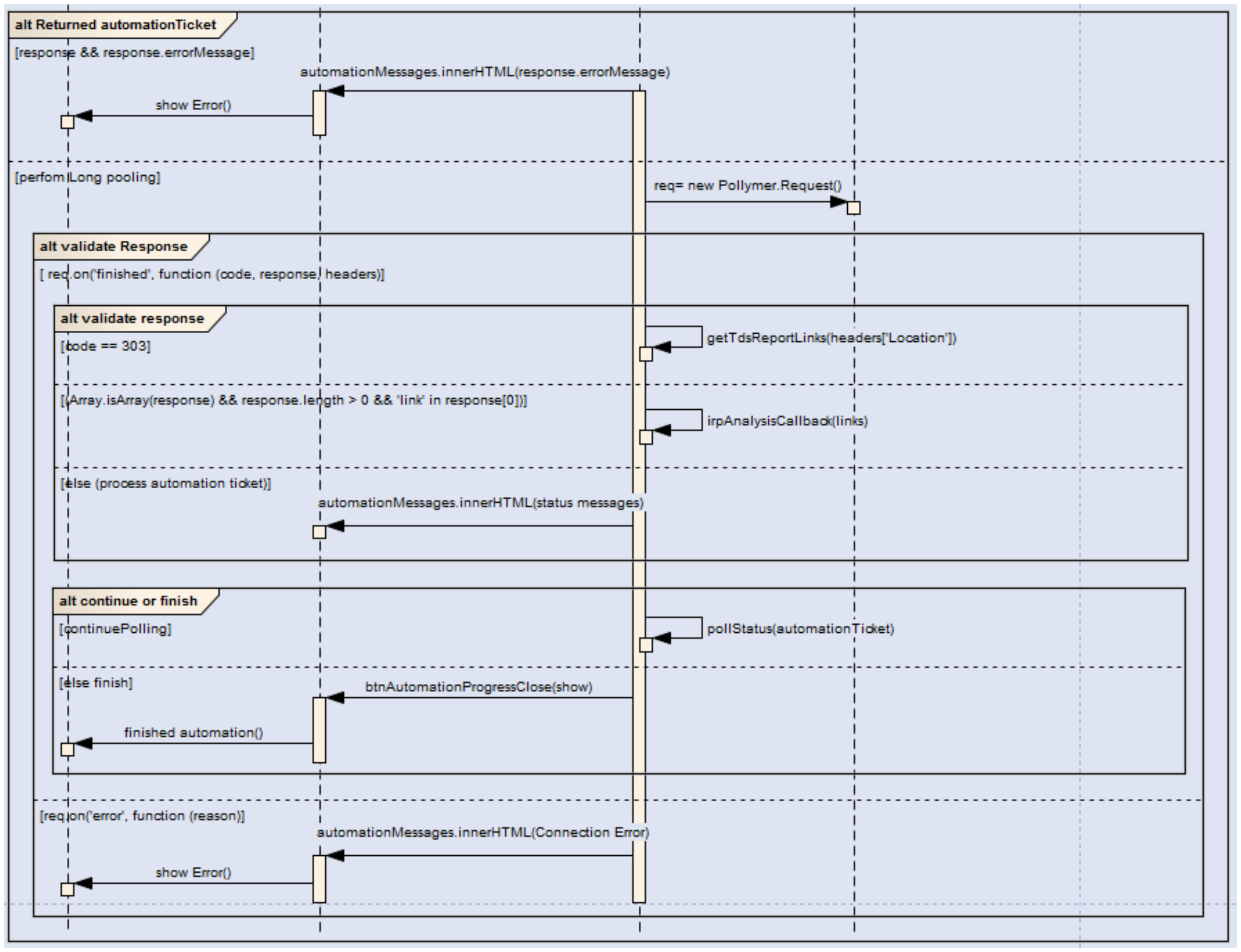
alt Returned automationTicket
[response && response.errorMessage]
automationMessages.innerHTML(response.errorMessage)
show Error()
[perfom Long pooling]
req= new Pollymer.Request()
alt validate Response
[req.on('finished', function (code, response, headers)]
alt validate response
[code == 303]
getTdsReportLinks(headers['Location'])
[(Array.isArray(response) && response.length > 0 && 'link' in response[0])]
irpAnalysisCallback(links)
[else (process automation ticket)]
automationMessages.innerHTML(status messages)
alt continue or finish
[continuePolling]
pollStatus(automationTicket)
[else finish]
btnAutomationProgressClose(show)
finished automation()
[req.on('error', function (reason)]
automationMessages.innerHTML(Connection Error)
show Error()

Figure 12. Diagram of the processing response.

1.2.1.4 Managing the AutomationTicket object

Validate Response.

There are two events that are triggered once the request is processed:

1) Req.on(finished).
   a. Process Response and AutomationTicket.
      i. Code 303. In this step the reponse is validated, if an HTTP code is equal to 303, that means the automation process finished and a call to the method getTdsReporLinks is performed.
      ii. Array.isArray(response) && response.length > 0 && 'link' in response[0]. Verify if the response contains an array, if the array length is more than zero, also if there is a link in the first item. If the validation is passed then the method irpAnalysisCallback() is invoked.
      iii. Process AutomationTicket. In this step the status messages are taken from this object and the UI is updated.
   b. Verify the Boolean variable "continuePooling"is true then.
      i. If is true, then an invocation to the pollStatus() method is performed again.
      ii. Else, the automation process is finished and the close button for the dialog open in the UI is displayed.
2) Req.on(error). If there is an error during the automation process, then this event is triggered, to finish all the process and display the error message to the user.

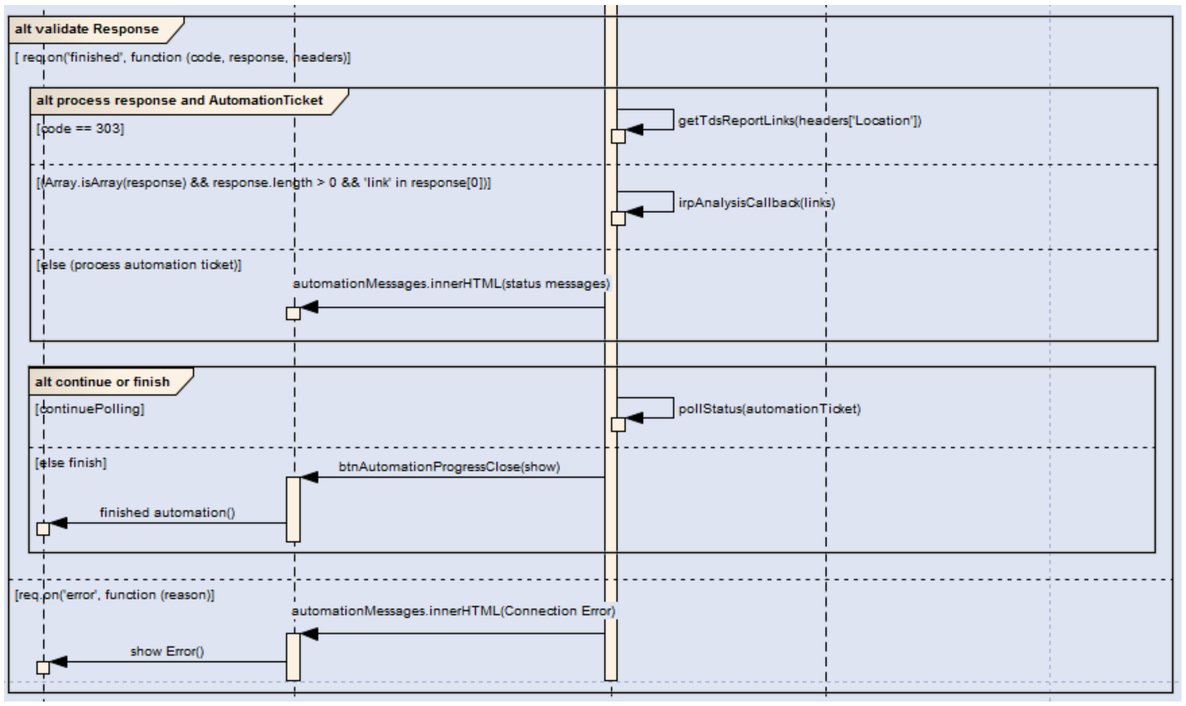
alt validate Response
[req.on('finished', function (code, response, headers)]
alt process response and AutomationTicket
[code == 303]
getTdsReportLinks(headers['Location'])
[!Array.isArray(response) && response.length > 0 && 'link' in response[0])]
irpAnalysisCallback(links)
[else (process automation ticket)]
automationMessages.innerHTML(status messages)
alt continue or finish
[continuePolling]
pollStatus(automationTicket)
[else finish]
btnAutomationProgressClose(show)
finished automation()
[req.on('error', function (reason)]
automationMessages.innerHTML(Connection Error)
show Error()

Figure 13. Diagram of the validation response.

1.2.2 Automation Process (Server Side).

The next diagrams describe the process involved in the automation process in the server side.

1.2.2.1 Start Automation process.

When the endpoint “/tdsReports” is requested using an HTTP POST method, the next process is executed in the server side.

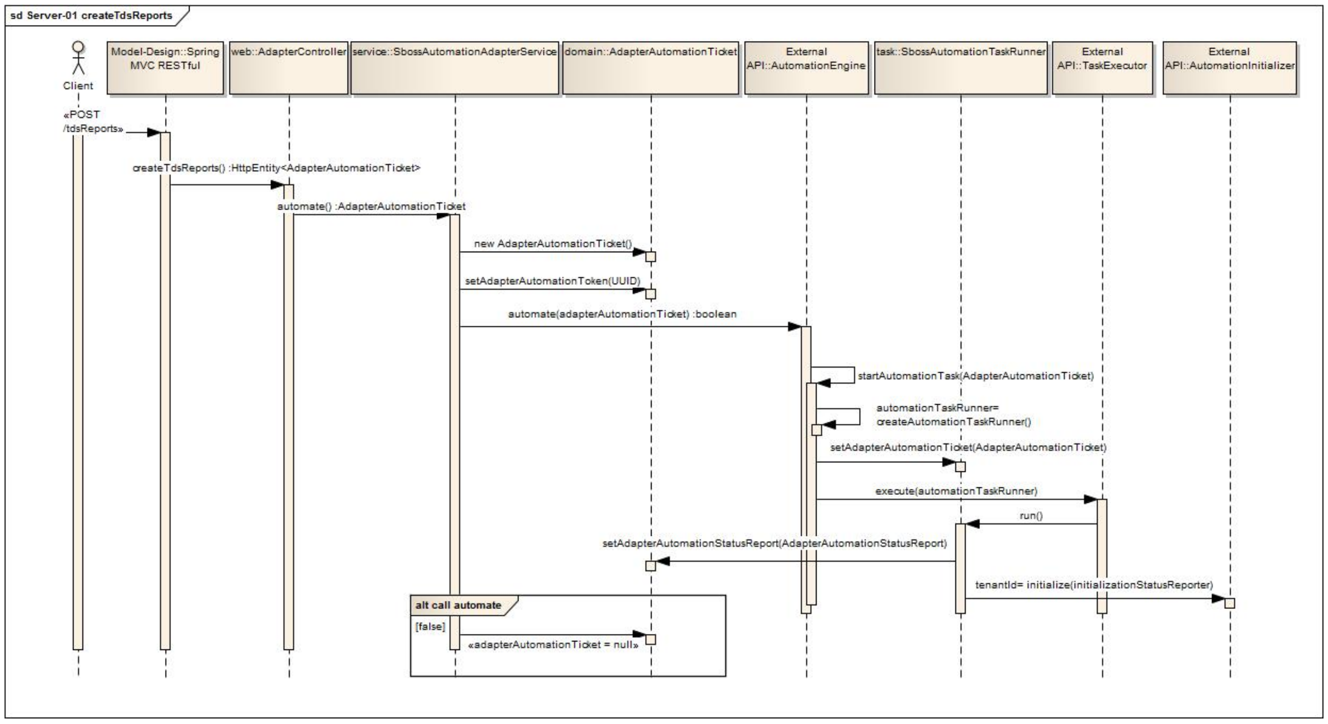
sd Server-01 createTdsReports
Client
Model-Design::Spring MVC RESTful
web::AdapterController
service::SbossAutomationAdapterService
domain::AdapterAutomationTicket
External API::AutomationEngine
task::SbossAutomationTaskRunner
External API::TaskExecutor
External API::AutomationInitializer
«POST /tdsReports»
createTdsReports() :HttpEntity<AdapterAutomationTicket>
automate() :AdapterAutomationTicket
new AdapterAutomationTicket()
setAdapterAutomationToken(UUID)
automate(adapterAutomationTicket) :boolean
startAutomationTask(AdapterAutomationTicket)
automationTaskRunner= createAutomationTaskRunner()
setAdapterAutomationTicket(AdapterAutomationTicket)
execute(automationTaskRunner)
run()
setAdapterAutomationStatusReport(AdapterAutomationStatusReport)
tenantId= initialize(initializationStatusReporter)
alt call automate
[false]
«adapterAutomationTicket = null»

Figure 13. Diagram of the automation process starting.

1.2.2.2 Obtaining the status of the current process.

When the endpoint "/tdsReports/queue/{adapterAutomationToken}" is requested using an HTTP GET method, the next process is executed in the server side.

The details about the process involved in this endpoint are described below.

1) The client request "/tdsReports/queue/{adapterAutomationToken}" passing as a path variable the token.
2) The process is received for the AdapterController object and the method getAutomationTicketStatus() method is executed
3) An implementation of Callable is created to manage a long polling. Callable allows the thread that received the call to be released, but the communication with the client is still open.
4) The getAdapterAutomationTicket() method is invoked to get the ticket, according to the token passed as parameter.
5) The next validation of the AdapterAutomationStatusReport is performed:
   a. If the the ticket has a status of complete, then another validation is performed:
      i. If status report has no error then an HttpHeaders object is created, and invoked the setLocation() method passing as parameter the URI of the current controller, in this case will be "/tdsReports" , then a new ResponseEntity object is created passing the httpHeaders object created before and HttpStatus.SEE_OTHER as parameters in the constructor.
      ii. A new ResponseEntity object is created passing the ticket object and an HttpStatus.INTERNAL_SERVER_ERROR as parameters in the constructor.
   b. A new ResponseEntity object is created passing the ticket object and an HttpStatus.INTERNAL_OK as parameters in the constructor

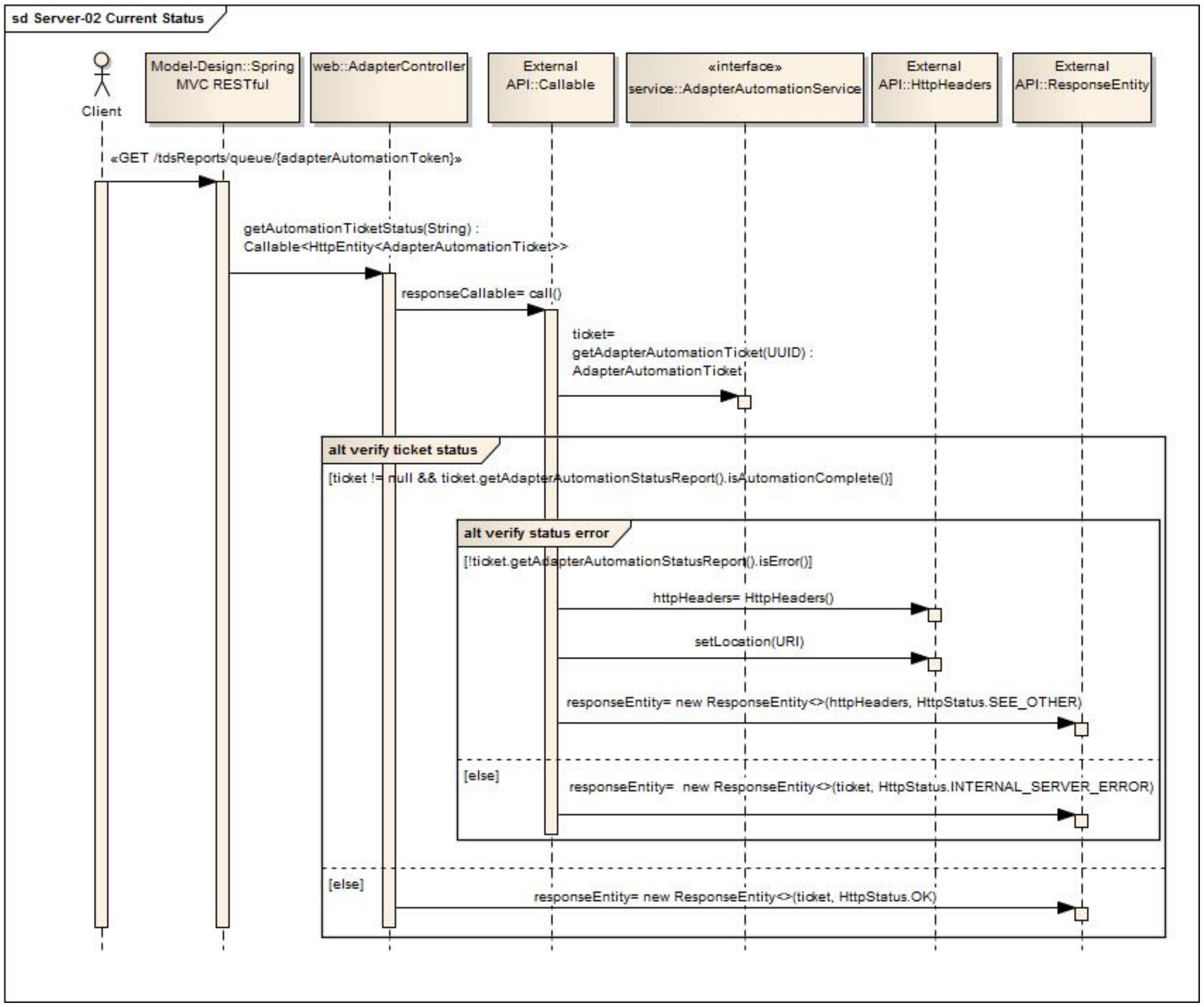
sd Server-02 Current Status
Client
Model-Design::Spring MVC RESTful
web::AdapterController
External API::Callable
«interface» service::AdapterAutomationService
External API::HttpHeaders
External API::ResponseEntity
«GET /tdsReports/queue/{adapterAutomationToken}»
getAutomationTicketStatus(String) : Callable<HttpEntity<AdapterAutomationTicket>>
responseCallable= call()
ticket= getAdapterAutomationTicket(UUID) : AdapterAutomationTicket
alt verify ticket status
[ticket != null && ticket.getAdapterAutomationStatusReport().isAutomationComplete()]
alt verify status error
[!ticket.getAdapterAutomationStatusReport().isError()]
httpHeaders= HttpHeaders()
setLocation(URI)
responseEntity= new ResponseEntity<>(httpHeaders, HttpStatus.SEE_OTHER)
[else]
responseEntity= new ResponseEntity<>(ticket, HttpStatus.INTERNAL_SERVER_ERROR)
[else]
responseEntity= new ResponseEntity<>(ticket, HttpStatus.OK)

Figure 13. Diagram of the request for getting the status of the current process.

1.2.2.3 Getting the Ids of the Reports.

When the endpoint "/tdsReports" is requested using an HTTP GET method, the next process is executed in the server side.

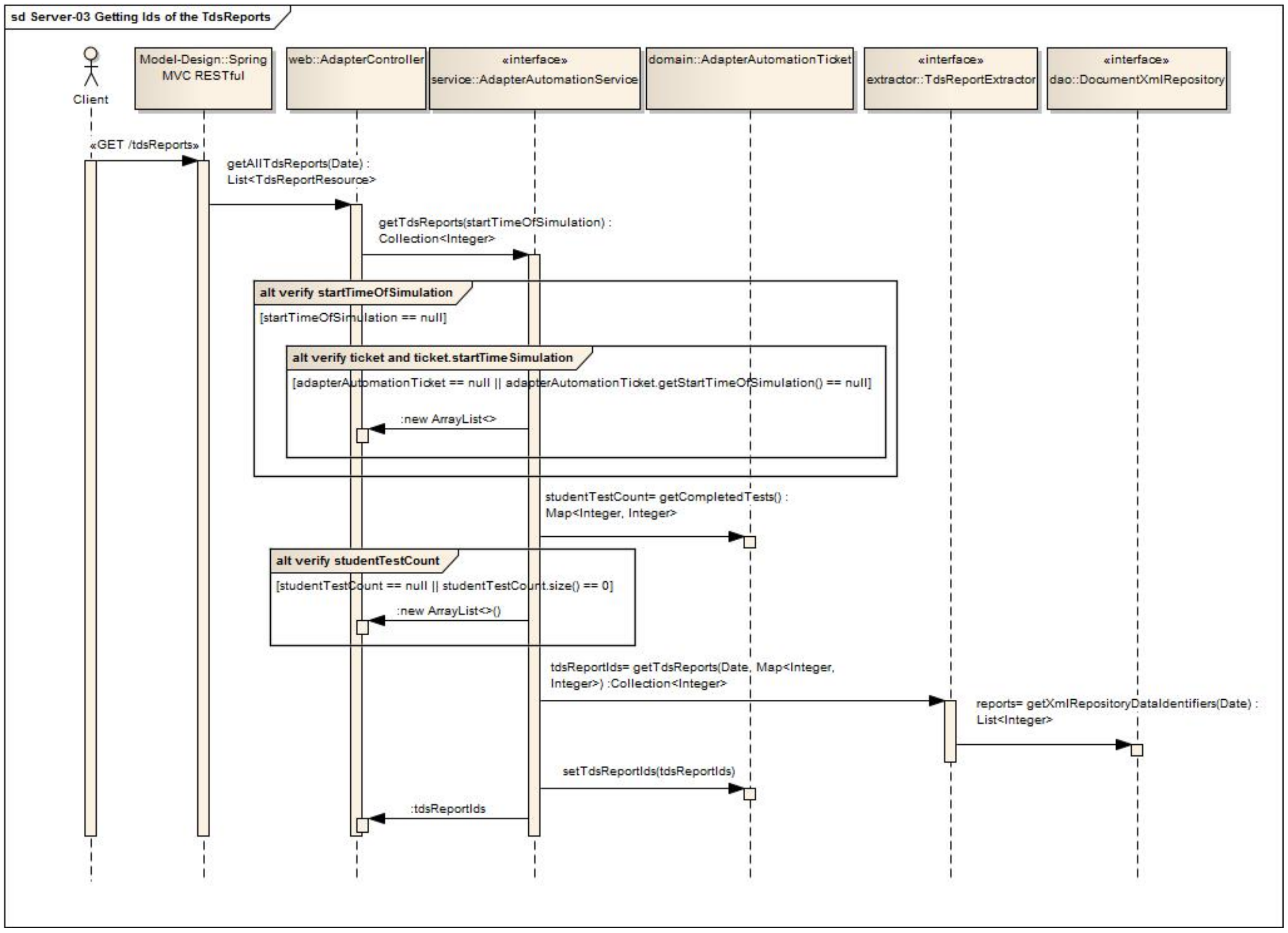
sd Server-03 Getting Ids of the TdsReports
Client
Model-Design::Spring MVC RESTful
web::AdapterController
«interface» service::AdapterAutomationService
domain::AdapterAutomationTicket
«interface» extractor::TdsReportExtractor
«interface» dao::DocumentXmlRepository
«GET /tdsReports»
getAllTdsReports(Date) : List<TdsReportResource>
getTdsReports(startTimeOfSimulation) : Collection<Integer>
alt verify startTimeOfSimulation
[startTimeOfSimulation == null]
alt verify ticket and ticket.startTime Simulation
[adapterAutomationTicket == null || adapterAutomationTicket.getStartTimeOfSimulation() == null]
:new ArrayList<>
studentTestCount= getCompletedTests() : Map<Integer, Integer>
alt verify studentTestCount
[studentTestCount == null || studentTestCount.size() == 0]
:new ArrayList<>()
tdsReportIds= getTdsReports(Date, Map<Integer, Integer>) :Collection<Integer>
reports= getXmlRepositoryDataIdentifiers(Date) : List<Integer>
setTdsReportIds(tdsReportIds)
:tdsReportIds

Figure 13. Diagram that shows the process to obtain the Ids of the reports generated during the automation process.

1.2.2.4 Getting XML Report.

When the endpoint “/tdsReports/{tdsReportId}” is requested using an HTTP GET method, the next process is executed in the server side.

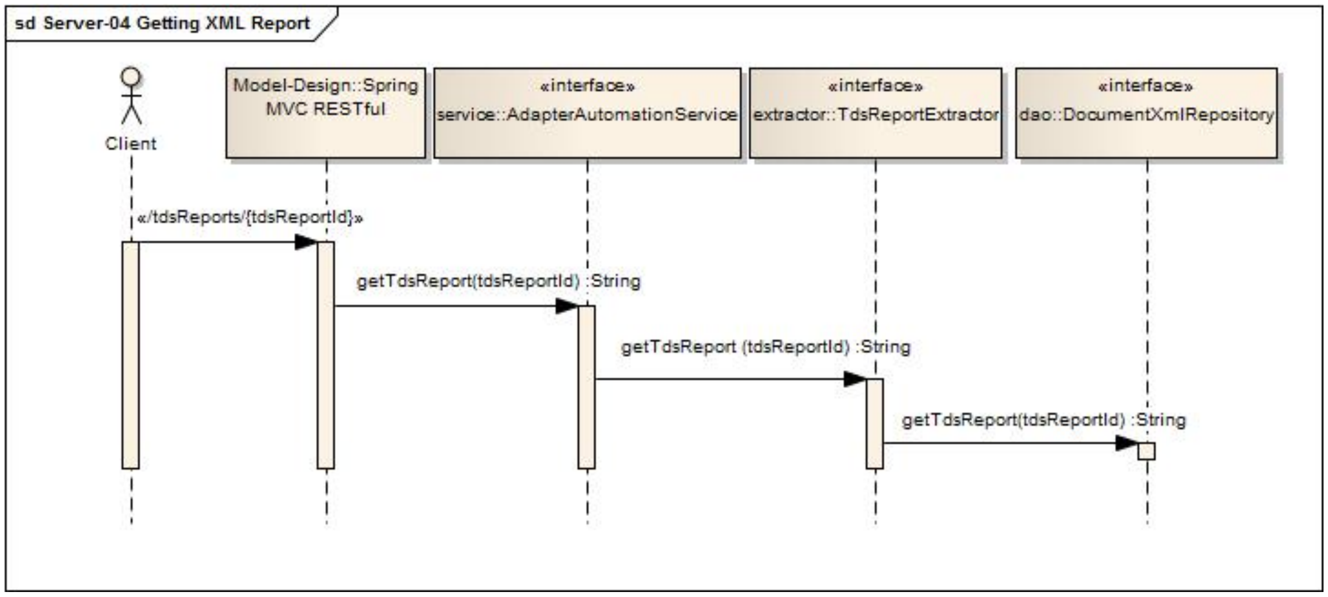
sd Server-04 Getting XML Report
Client
Model-Design::Spring MVC RESTful
«interface» service::AdapterAutomationService
«interface» extractor::TdsReportExtractor
«interface» dao::DocumentXmlRepository
«/tdsReports/{tdsReportId}»
getTdsReport(tdsReportId) :String
getTdsReport (tdsReportId) :String
getTdsReport(tdsReportId) :String

Figure 13. Diagram that shows the process to get the XML report according to the ID.

1.3 RESTful service.

1.3.1 Documentation

Name	**Description**
Title	Start Automation
Description	Initializes the automation process, and returns the location of the next service to be called.
URL	/tdsReports
Method	POST
URL Params	N/A
Data Params	N/A

Success Response	`Headers:`
	
	`Location- This header contains the next URL to be redirected. Example: Location=[/tdsReports/queue/16f7d0a0-9b74-4ceb-9512-c052660960a0]`
	
	`Object: AdapterAutomationTicket`
	
	HttpStatus: ACCEPTED

Notes	This endpoint is the first one to be called, so the automation can start.

Name	**Description**
Title	Status of the automation process.
Description	Gets the latest update of the AdapterAutomationTicket for the given Automation Token.
URL	/tdsReports/queue/{adapterAutomationToken}
Method	GET
URL Params	N/A.
Data Params	N/A
Success Response	**Response 1:** Object: AdapterAutomationTicket HttpStatus: OK **Response 2:** Headers: Location- This header contains the next URL to be redirected. Example: Location= [/tdsReports/] HttpStatus: SEE_OTHER
Error Response	Object: AdapterAutomationTicket HttpStatus: INTERNAL_SERVER_ERROR
Notes	

Name	**Description**
Title	Report IDs
Description	Gets all of the generated Ids of the TDSReports starting from the given date and time or since the time of previous test Simulation.
URL	/tdsReports
Method	GET
URL Params	N/A
Data Params	N/A
Success Response	Returns resource links such as :http://<server>:<port>/tdsReports/<id defined above>
Notes	This endpoint is called once the automation process is finished.

Name	**Description**
Title	XML Report
Description	Gets an individual TDSReport, according to the tdsReportId passed as parameter
URL	/tdsReports/{tdsReportId}
Method	GET
URL Params	N/A
Data Params	N/A
Success Response	Returns a text, containing an XML document.
Notes	

1.4 Revision History.

Date	Version	Description	Author
Feb/23/2017	1.0	Initial document	Ernesto De La Luz