Web Application Communication

Case: Automotive Connectivity Platform Kanzi Connect

Lasse Malmberg

Master's thesis
August 2021
Information and communication technologies
Master's Degree Programme in Information Technology
Full Stack Software Development

jamk | Jyväskylän ammattikorkeakoulu
University of Applied Sciences

jamk

Description

Malmberg, Lasse

Web Application Communication, Case: Automotive Connectivity Platform Kanzi Connect

Jyväskylä: JAMK University of Applied Sciences, August 2021, 74 pages.

Information and communication technologies. Master’s Degree Programme in Information Technology, Full Stack Software Development. Master’s Thesis.

Permission for web publication: Yes

Language of publication: English

Abstract

Web application communication was explored in the context of the automotive connectivity platform Kanzi Connect. The development task was commissioned by Rightware, which is a company that develops user interface and connectivity technology for the automotive industry. The objective was to improve web-based communication with a Kanzi Connect server.

Web application communication was examined from the perspective of API design and communication protocols. Web-based communication with a Kanzi Connect server is particularly important to its web user interface Kanzi Connect Simulator. While the results are generally applicable, the focus was specifically on aspects of web application communication that are relevant to Kanzi Connect Simulator.

Web-based communication with a Kanzi Connect server was improved by implementing a WebSocket API. Kanzi Connect Simulator was improved in terms of web application communication by implementing new features and solving existing problems using the WebSocket API. The Kanzi Connect HTTP API was also evaluated.

Keywords/tags (subjects)

Web Application Communication, Web-Based Communication, WebSocket, RPC, REST, API, API Design, Automotive, Rightware, Kanzi, Kanzi Connect

Miscellaneous (Confidential information)

References to Rightware’s online documentation (docs.kanzi.com) require authentication to access.

Contents

1 Introduction ... 5
2 Research Setting ... 5
3 Background ... 7
3.1 Rightware ... 7
3.2 Kanzi UI ... 8
3.3 Kanzi Connect ... 9
3.3.1 Fundamentals ... 10
3.3.2 Services ... 11
3.3.3 HTTP API ... 13
3.4 Kanzi Connect Simulator ... 14
3.4.1 Services ... 14
3.4.2 Scenarios ... 15
3.5 Workflow ... 17
4 Objectives ... 19
4.1 State of Kanzi Connect Server ... 20
4.2 Service Data Updates ... 22
4.3 Service Event Notifications ... 23
4.4 Improving the HTTP API ... 24
5 Browser Networking ... 24
5.1 HTTP ... 24
5.2 XHR ... 27
5.3 Fetch ... 28
5.4 SSE ... 28
5.5 WebSocket ... 30
5.6 WebRTC ... 31
6 Web APIs ... 31
6.1 API Design ... 31
6.2 RPC APIs ... 33
6.3 REST APIs ... 34
6.4 GraphQL APIs ... 38
6.5 OpenAPI ... 40
7 Development ... 42
7.1 WebSocket API ... 42

7.1.1 Server-Side Implementation 42
7.1.2 Client-Side Implementation 50
7.2 HTTP API 54
7.2.1 Overview 54
7.2.2 Service API 57
7.2.3 Scenario API 59
8 Output 62
9 Ethics 63
10 Discussion 65
11 Conclusion 70
References 71

Figures

Figure 1. Kanzi Reference HMI 8
Figure 2. Kanzi Studio 9
Figure 3. Kanzi Connect server 10
Figure 4. Kanzi Connect network 11
Figure 5. Service interface definition 12
Figure 6. Service code generation 13
Figure 7. Services view in Kanzi Connect Simulator 15
Figure 8. Scenarios view in Kanzi Connect Simulator 16
Figure 9. Kanzi Studio, Kanzi Connect server, and Kanzi Connect Simulator 17
Figure 10. Creating a new service in Kanzi Connect Simulator 17
Figure 11. Creating a new service data element in Kanzi Connect Simulator 18
Figure 12. Importing a Kanzi Connect service to Kanzi Studio 19
Figure 13. Server state indicator in Kanzi Connect Simulator 20
Figure 14. State synchronization in Kanzi Connect Simulator 20
Figure 15. Slider controller in Kanzi Connect Simulator 21
Figure 16. Cypress end-to-end test framework user interface 22
Figure 17. Service data controllers in Kanzi Connect Simulator 23
Figure 18. Service events in Kanzi Connect Simulator 24
Figure 19. API goals canvas 33
Figure 20. Richardson maturity model 37
Figure 21. OpenAPI definition preview in Visual Studio Code 41
Figure 22. WebSocket plugin in Kanzi Connect server configuration file 42

Figure 23. WebSocket plugin handler classes 43
Figure 24. CivetWebSocketHandler class 44
Figure 25. RuntimedataWebSocketHandler class....45
Figure 26. JSON message sent by RuntimedataWebSocketHandler....47
Figure 27. EventWebSocketHandler class 47
Figure 28. JSON message sent by EventWebSocketHandler 49
Figure 29. Cluster service runtime data objects in Kanzi Connect Simulator 51
Figure 30. Settings panel in Kanzi Connect Simulator 52
Figure 31. Service register and unregister notifications in Kanzi Connect Simulator 53
Figure 32. User interface responsiveness in Kanzi Connect Simulator 67
Figure 33. Diagnostics view in Kanzi Connect Simulator 68
Figure 34. Version 2.1 of Kanzi Connect Simulator....69

Code Blocks

Code Block 1. Invoking service methods using the HTTP API 13
Code Block 2. HTTP API response to service method invocation 14
Code Block 3. HTTP API endpoints for querying services 14
Code Block 4. HTTP request sent by Kanzi Connect Simulator 25
Code Block 5. HTTP response received by Kanzi Connect Simulator 26
Code Block 6. XMLHttpRequest example 27
Code Block 7. Fetch example 28
Code Block 8. Server-sent events client-side example 29
Code Block 9. Server-sent events server-side example 29
Code Block 10. WebSocket client-side example....30
Code Block 11. Slack RPC Web API 34
Code Block 12. GitHub REST API request and response example....38
Code Block 13. GitHub GraphQL API request and response example 39
Code Block 14. OpenAPI definition example....40
Code Block 15. WebSocket handler registration 44
Code Block 16. Managing WebSocket handler connections 46
Code Block 17. Sending runtime data changes to clients....46
Code Block 18. Subscribing to service events....48
Code Block 19. Sending events to clients 49
Code Block 20. Handling WebSocket runtime data updates 50

Code Block 21. Handling WebSocket events 54
Code Block 22. Persistence service HTTP API endpoints 55
Code Block 23. Extended service definition and the resulting endpoints 56
Code Block 24. ServiceManager service API for working with services 57
Code Block 25. Modified ServiceManager service API for working with services......................58
Code Block 26. ServiceManager service API for service event elements..................................58
Code Block 27. Scenario service API for working with scenarios..60
Code Block 28. Scenario service API for working with scenario components 61

1 Introduction

The thesis explores web application communication in the context of the automotive connectivity platform Kanzi Connect. The thesis is commissioned by Rightware, which is a company that develops user interface and connectivity technology for the automotive industry. The objective of the thesis is to improve web-based communication with a Kanzi Connect server. The thesis examines web application communication from the perspective of API design and communication protocols. Web-based communication with a Kanzi Connect server is particularly important to its web user interface Kanzi Connect Simulator. While the results of the thesis are generally applicable, the thesis aims to specifically improve aspects of web application communication that are relevant to Kanzi Connect Simulator.

The thesis starts by presenting the research setting and the research method. The Kanzi software is then introduced to the degree that is required to understand the research objectives in the following chapter. The theoretical basis covers both browser networking and Web APIs. The chapter on browser networking investigates the different APIs and protocols that are available to web applications in web browsers. Web APIs are examined from the perspective of API design guidelines and three different API styles: RPC, REST, and GraphQL. The development chapter implements solutions to the research problems based on the theoretical basis. The output of the development is then analyzed followed by ethical considerations. The thesis concludes with a discussion on the results, applicability, and future of the research and development.

2 Research Setting

The theoretical research objective is to gain a broad understanding of web application communication. The practical research objective is to improve web-based communication with a Kanzi Connect server. The primary research question of the thesis is "How can web-based communication with a Kanzi Connect server be improved?". Two secondary research questions derive from the primary research question: "How can the existing Kanzi Connect HTTP API be improved?" and "How can Kanzi Connect Simulator be improved in terms of web application communication?".

The thesis uses the master branch of Kanzi Connect and version 2.0 of Kanzi Connect Simulator as the baseline for evaluation and development. The master branch of Kanzi Connect currently serves

as the development branch for Kanzi Connect version 2.0. Kanzi Connect server-side code is developed using C++. Kanzi Connect Simulator is developed using JavaScript and the Vue.js framework.

Web application communication is only evaluated in the context of a development environment. Production use of Kanzi Connect is outside the scope of the thesis. This is because Kanzi Connect Simulator is a development tool that is only used in a development environment. As a result, certain topics such as security are not covered in depth.

The thesis uses the constructive research method which aims to produce innovative constructs to solve real-world problems. A construct is invented and developed and has many possible realizations. Man-made artifacts such as information system models and mathematical algorithms are examples of constructs. (Lukka 2014.)

The main attributes of the constructive research method are that it:

- focuses on real-world problems that are considered meaningful to solve
- produces an innovative construct which is meant to solve the original problem and attempts the implementation of the construct in practice to test its suitability
- includes close collaboration between the researcher and the practical stakeholders where experiential learning is expected to happen
- is carefully tied to existing theoretical knowledge
- takes special care to reflect empirical findings back to theory (Lukka 2014)

Constructive research is experimental in nature: the developed and implemented new construct should be viewed as an instrument which is used to demonstrate, test, and refine existing theory or to create a new theory altogether. The constructive research methodology is based on an idea originating from pragmatism where a thorough practical analysis of what works or does not work can be used to produce substantial theoretical contributions. The ideal result of constructive research is that the real-world problem is solved with the implemented new construct and that the problem-solving process produces a large contribution both from a theoretical and a practical perspective. (Lukka 2014.)

The constructive research methodology typically uses a research process that consists of the following steps:

1. Find a practically relevant problem that includes a possibility for theoretical contribution.
2. Investigate whether there is a possibility for long-term research collaboration with the target organization.
3. Obtain a deep understanding of the research topic both practically and theoretically.
4. Innovate a solution model and develop a construct that solves the problem, and which might also have potential for a theoretical contribution.
5. Implement the solution and test its functionality.
6. Consider the applicability of the solution.
7. Identify and analyze the theoretical contribution (Lukka 2014).

3 Background

This chapter covers the background knowledge needed to understand the objectives of the thesis. It introduces Rightware as a company and explains its products to the degree that is relevant to understand the context and requirements for web application communication in Kanzi Connect and Kanzi Connect Simulator.

3.1 Rightware

Rightware is a software company that provides tools and services for development of advanced digital user interfaces. The company was founded in 2009 and is headquartered in Finland. Rightware was acquired in 2016 by the Chinese software company ThunderSoft. Kanzi software is in production use by over 50 automotive brands and is expected to power the user experience in over 40 million cars by 2024. (Rightware 2016; Rightware 2021a; Rightware 2021f; Rightware 2021i.)

Rightware's main products are Kanzi UI, Kanzi Connect, and Kanzi Reference HMI. Kanzi UI enables user interfaces to be quickly designed and developed for the automotive industry and other embedded applications. Kanzi Connect is a highly customizable connectivity platform for sharing content and services. Kanzi Reference HMI is a starter kit for modern human-machine interface development using the Kanzi software tools. Rightware's main products are complemented by solutions such as Kanzi Maps, Kanzi Particles, and Kanzi Safety. Figure 1 shows Kanzi Reference HMI in action. (Rightware 2021b; Rightware 2021c; Rightware 2021d; Rightware 2021e; Rightware 2021g; Rightware 2021j.)

Figure 1. Kanzi Reference HMI

3.2 Kanzi UI

Kanzi UI is a solution for design and development of 2D and 3D user interfaces for the automotive industry and other embedded applications. Kanzi UI consists of two main components: Kanzi Studio and Kanzi Runtime. Kanzi Studio is a real-time UI editor with live preview. Kanzi Runtime is an industrial grade cross-platform graphics rendering engine. Kanzi Runtime runs user interfaces developed with Kanzi Studio on the target platform. Kanzi Studio is shown in Figure 2. (Rightware 2021c; Rightware 2021h.)

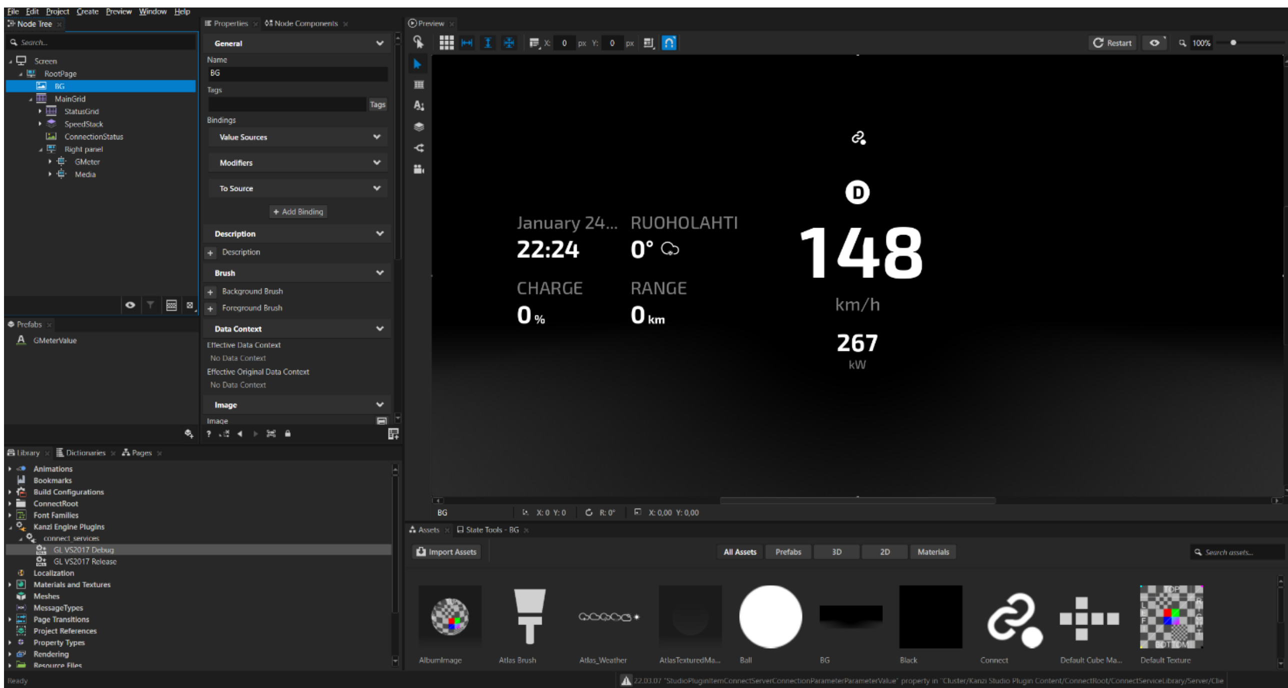

Figure 2. Kanzi Studio

3.3 Kanzi Connect

Kanzi Connect is a highly customizable connectivity platform that allows sharing of data, assets, and services between multiple clients. It is used to add connectivity features to Kanzi applications. Kanzi Connect uses a client-server-based architecture and enables creating and managing of complex multi-device setups. (Kanzi Documentation 2021a; Rightware 2021d.) Figure 3 illustrates the Kanzi Connect server.

C:/code/connect/runtime/lib/win32/GL_vs2017_Debug_DLL\connect_server.exe

```
2021-07-27T09:08:30+0300 14452 INF [              secmanager] OpenSSL version OpenSSL 1.1.0g  2 Nov 2017
2021-07-27T09:08:30+0300 14452 WRN [              secmanager] Could not load PKCS#12 file in C:\code\connect\runtime/../
configs/../certs/masterca.pfx (error:0D07209B:asn1 encoding routines:ASN1_get_object:too long)
2021-07-27T09:08:30+0300 14452 INF [              secmanager] Using master CA certificate from C:\code\connect\runtime/.
./configs/../certs/master_ca.crt.pem
2021-07-27T09:08:30+0300 19712 INF [      work_queue_private] Worker thread starting.
2021-07-27T09:08:30+0300 17788 INF [      work_queue_private] Worker thread starting.
2021-07-27T09:08:30+0300 11072 INF [      work_queue_private] Worker thread starting.
2021-07-27T09:08:30+0300 14452 INF [           connectdomain] Registered metaclass 7C16A068 from module: policyplugin_de
fault
2021-07-27T09:08:30+0300 14452 INF [           connectdomain] Registered module with name policyplugin_default
2021-07-27T09:08:30+0300 14452 INF [           policycontext] PolicyContext initialization postponed because no domain i
nstance was given
2021-07-27T09:08:30+0300 14452 INF [           policycontext] Mode = Notify
2021-07-27T09:08:31+0300 14452 INF [           policycontext] Policies loaded from policy.yaml. Mode is Notify.
2021-07-27T09:08:31+0300 14452 INF [                  server] Initializing civet server
2021-07-27T09:08:31+0300 14452 INF [                  server] Finished civet initialization successfully
2021-07-27T09:08:31+0300 14452 INF [     virtualfile_service] Found prefix: 'http://<server>:8080'
2021-07-27T09:08:31+0300 14452 INF [     virtualfile_service] Found prefix: 'https://<server>:8843'
2021-07-27T09:08:31+0300 14452 INF [           connectdomain] Registered metaclass 7C24BE14 from module: websocketplugin
_default
2021-07-27T09:08:31+0300 14452 INF [           connectdomain] Registered module with name websocketplugin_default
2021-07-27T09:08:31+0300 14452 INF [         service_manager] Service Connect.Service.ServiceInvoke instantiated.
2021-07-27T09:08:31+0300 14452 INF [         service_manager] Service Connect.Service.Cluster instantiated.
2021-07-27T09:08:31+0300 14452 INF [         service_manager] Service Connect.Service.System instantiated.
2021-07-27T09:08:31+0300 14452 INF [         service_manager] Service Connect.Service.Media instantiated.
2021-07-27T09:08:31+0300 14452 INF [         service_manager] Service Connect.Service.Sensor instantiated.
2021-07-27T09:08:31+0300 14452 INF [         service_manager] Service Connect.Service.Obd2 instantiated.
```

Figure 3. Kanzi Connect server

3.3.1 Fundamentals

Kanzi Connect builds a Kanzi Connect network using a client-server architecture. Figure 4 illustrates a Kanzi Connect network. The network has a Kanzi Connect server and clients. The server hosts services which the clients consume. A service has an interface that defines a contract between a client and the rest of the Kanzi Connect system. The interface most commonly includes a data model. Kanzi Connect SDK includes tools for creating, modifying, and simulating services. (Kanzi Documentation 2021b.)

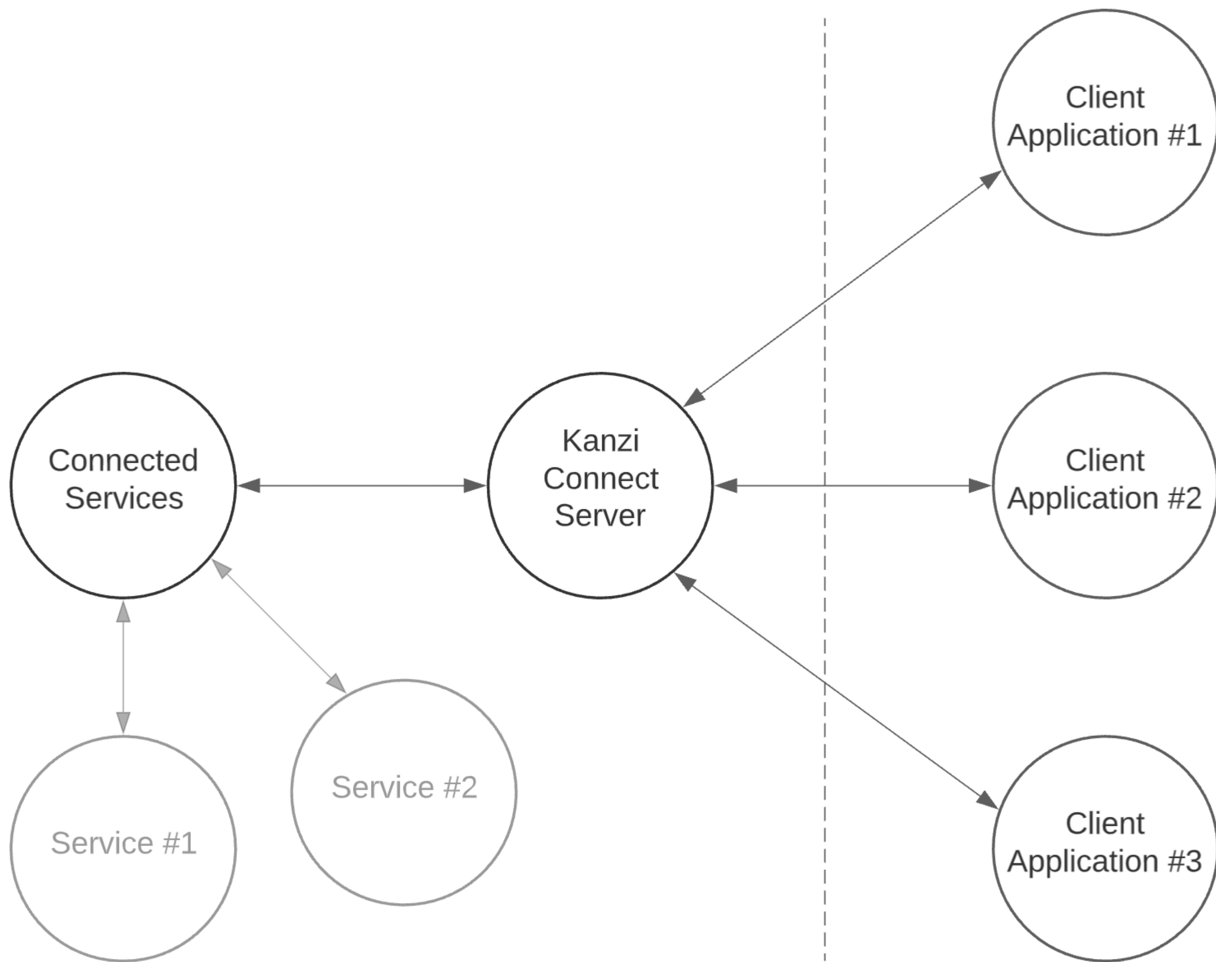

Figure 4. Kanzi Connect network

3.3.2 Services

Services add functionality to a Kanzi Connect server. The Kanzi Connect SDK comes with built-in services for common use cases such as content distribution, diagnostics, and data persistence. Kanzi Connect has a set of core services which are hard dependencies of the Kanzi Connect server and cannot be removed. Examples of core services include the connection service and the configuration service. Other services can be added and removed on demand. Services can be used to integrate with operating system and platform features, interface with external components, and synchronize the state of Kanzi Connect clients. Services can collect, process, and produce data for Kanzi Connect clients. (Kanzi Documentation 2021e; Kanzi Documentation 2021f.)

A service exposes an interface to the Kanzi Connect network which clients can access. The interface is defined in XML format and contains definitions for elements such as service data, methods, and events. Figure 5 shows an example of a service interface definition. (Kanzi Documentation 2021e.)

```xml
<service fullname="Connect.Service.Media" name="Media" description="Media playback interface">
    <event name="progress" description="Notification of playback state change">
        <argument datatype="int" name="position" default="0" />
        <argument datatype="int" name="duration" default="0" />
    </event>
    <method name="play" description="Continues playback of current track"/>
    <method name="pause" description="Pauses playback of current track"/>
    <method name="stop" description="Stops playback of current track"/>
    <runtime-data>
        <playback>
            <state datatype="int" default="0"/>
            <position datatype="string" default="00:00"/>
            <duration datatype="string" default="00:00"/>
            <offset datatype="float" default="0.000000"/>
            <shuffle datatype="bool" default="false"/>
        </playback>
        <current_track>
            <id datatype="int" default="0"/>
            <name datatype="string" default=""/>
            <artist datatype="string" default=""/>
            <album_id datatype="int" default="0"/>
            <uri datatype="string" default=""/>
            <image datatype="string" default=""/>
            <source datatype="string" default=""/>
        </current_track>
    </runtime-data>
</service>
```

Figure 5. Service interface definition

A service interface definition file is run through code generation scripts which produce stub files for the service. This includes both server-side stubs that the server implements and client-side stubs

that a client implements to interface with the server-side implementation. Figure 6 shows this process. (Kanzi Documentation 2021e.)

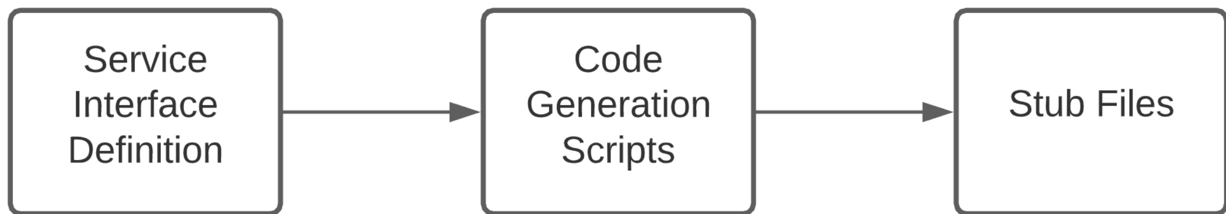

Figure 6. Service code generation

3.3.3 HTTP API

Kanzi Connect has a stateless HTTP API that provides access to service methods and data. Kanzi Connect embeds a web server called CivetWeb which together with the Virtual File core service enable HTTP support. The CivetWeb web server also hosts the Kanzi Connect Simulator web application. The HTTP API supports GET and POST HTTP methods. Code block 1 shows the format of service method invocation and an example invoking the play method of a media service. (Kanzi Documentation 2021c.)

```
// Format of service method invocation.
POST /serviceinvoke/{service}/method/{method}

// Example of service method invocation.
POST /serviceinvoke/media/method/play
```

Code Block 1. Invoking service methods using the HTTP API

The HTTP API responds to method invocations with a JSON document that describes the result of the invocation. Code block 2 shows an example of a response. (Kanzi Documentation 2021c.)

```json
{
    "status": "OK",
    "type": "int",
    "value": "0"
}
```

Code Block 2. HTTP API response to service method invocation

In addition to access to service methods, the HTTP API provides endpoints for querying data such as service interface definitions and service runtime data. Code block 3 shows examples of these endpoints. (Kanzi Documentation 2021c.)

```
// Get service runtime data values.
GET /serviceruntimedata/{service}

// Get service description.
GET /servicedescriptions/{service}
```

Code Block 3. HTTP API endpoints for querying services

3.4 Kanzi Connect Simulator

Kanzi Connect Simulator is a web user interface for a Kanzi Connect server. Its primary audience are designers and developers who work with Kanzi Connect. Simulator is a development tool, and it is not used in a production environment. Simulator has two main views: services and scenarios. The services view gives the user full control of services running on a Kanzi Connect server. The scenarios view builds on top of services by allowing the user to control services with JavaScript scripts. (Kanzi Documentation 2021d.)

3.4.1 Services

The services view is the most important part of the simulator. It enables the user to create, delete, and modify services running on a Kanzi Connect server in real-time. The user can also control services by invoking service methods, triggering service events, and changing the state of service runtime data. Figure 7 shows the services view. (Kanzi Documentation 2021d.)

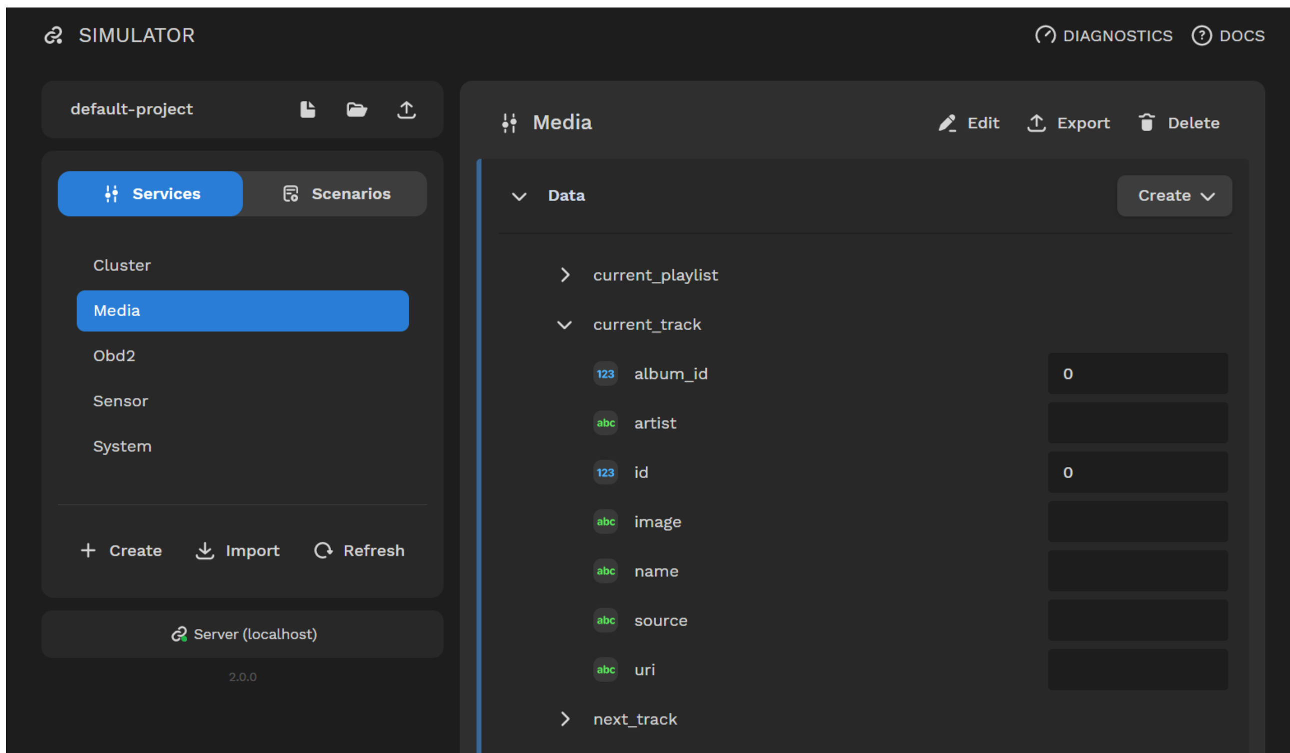

Figure 7. Services view in Kanzi Connect Simulator

3.4.2 Scenarios

The scenarios view builds on top of services by allowing the user to control services with JavaScript scripts. Figure 8 shows the scenarios view. A scenario consists of one or more scripts, which all have one or more triggers that determine when the script is executed. There are three types of triggers: data triggers, event triggers, and timer triggers. Data triggers set off in response to service runtime data changes, event triggers set off in response to service events, and timer triggers set off with a delay at time intervals. (Kanzi Documentation 2021g.)

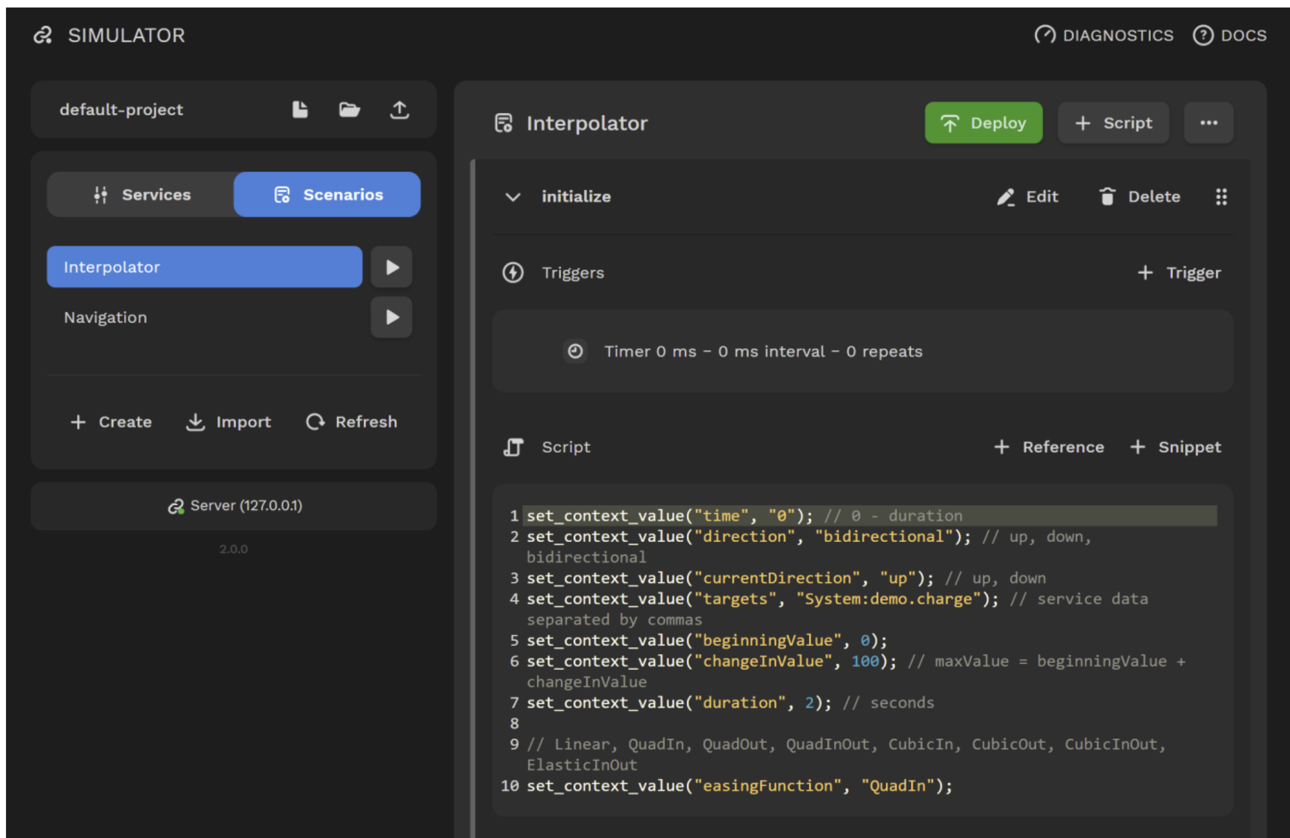

Figure 8. Scenarios view in Kanzi Connect Simulator

Scenarios are deployed to the Kanzi Connect server which executes the JavaScript using an embedded JavaScript engine called Duktape. The server implements a custom JavaScript API that allows the scripts to interface with the Kanzi Connect server. This allows scenarios to for example change the state of service runtime data and invoke service methods.

3.5 Workflow

This chapter explains the basic end user workflow of a developer or designer working with Kanzi Studio, Kanzi Connect, and Kanzi Connect Simulator. Figure 9 shows how Kanzi Studio, Kanzi Connect, and Kanzi Connect Simulator relate to each other.

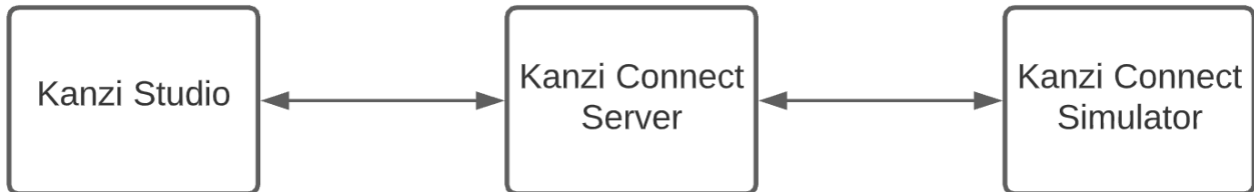

Figure 9. Kanzi Studio, Kanzi Connect server, and Kanzi Connect Simulator

The basic building block of functionality in Kanzi Connect is a service. Kanzi Connect Simulator can be used to create simulated services on the Kanzi Connect server. Figure 10 shows the modal window for creating a new service in simulator.

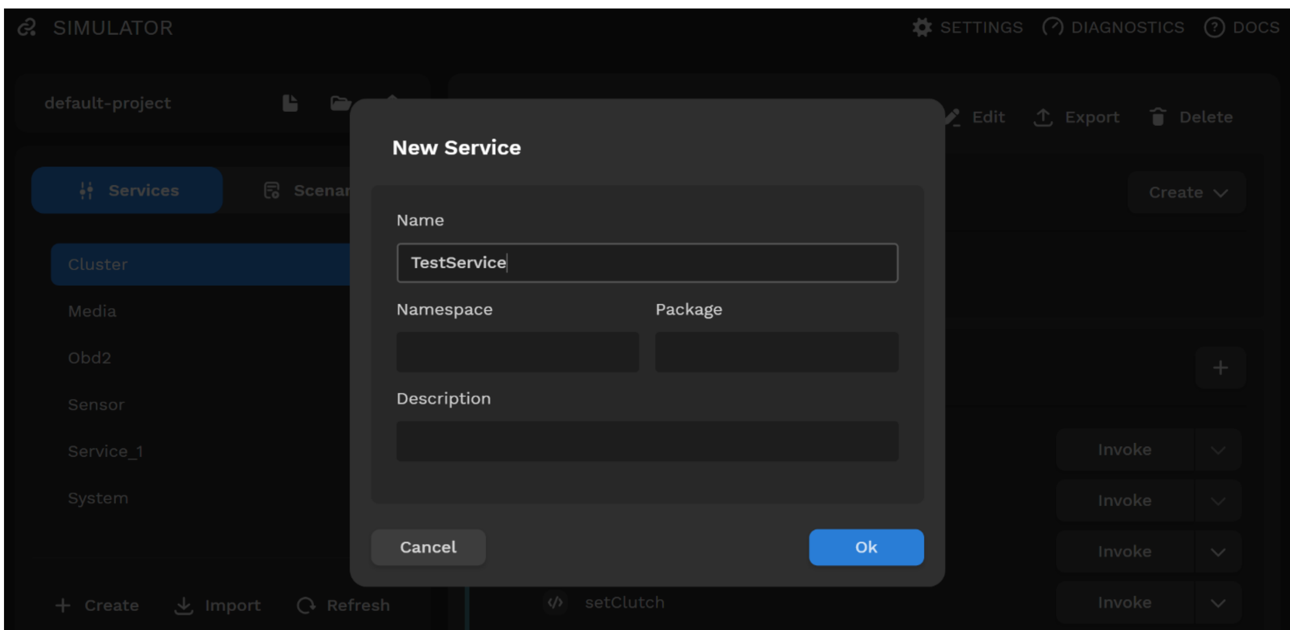

Figure 10. Creating a new service in Kanzi Connect Simulator

The structure and interface of a service can be quickly iterated on using simulator. All changes to a service in simulator update in real-time to the Kanzi Connect server, and from the server to Kanzi Studio. Figure 11 shows the modal window for adding a new data element to a service in simulator.

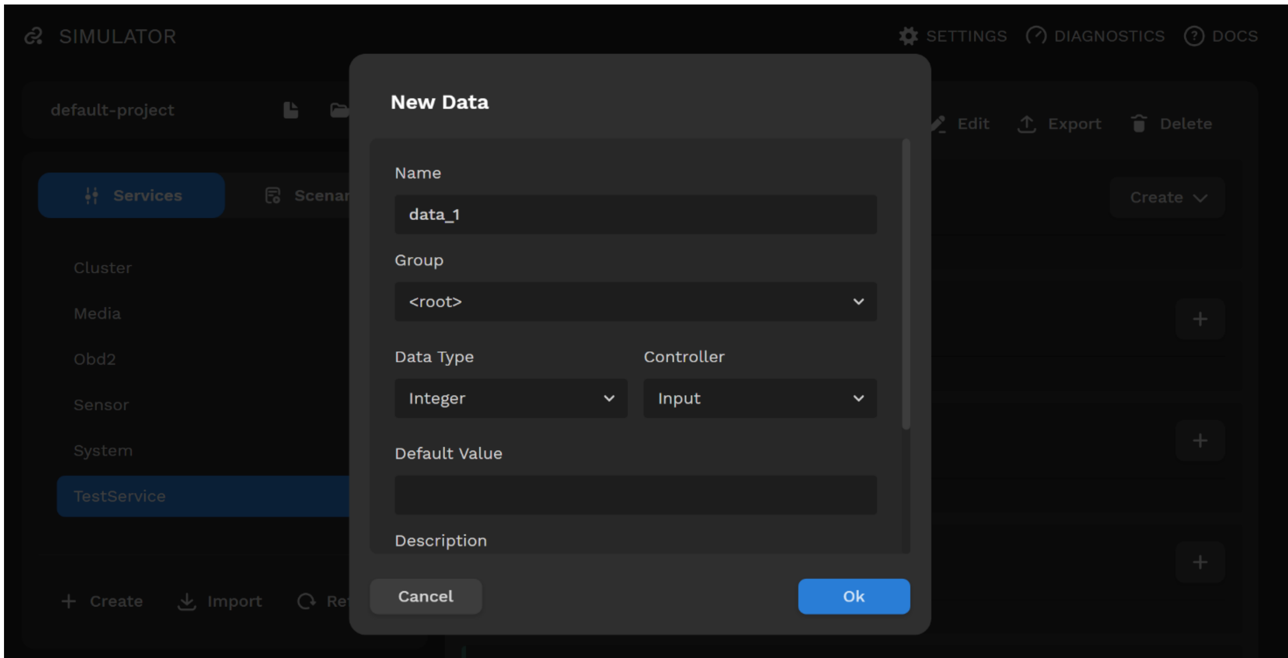

Figure 11. Creating a new service data element in Kanzi Connect Simulator

To integrate a service to a Kanzi user interface the service must be imported to Kanzi Studio using the Kanzi Connect tools for Kanzi Studio. Figure 12 shows the window for importing services in Kanzi Studio. Once imported, elements of a service can be bound to user interface elements. For example,

a button can be configured to invoke a service method on the Kanzi Connect server, or a text node can be bound to a service data element to display the state of service data in the user interface.

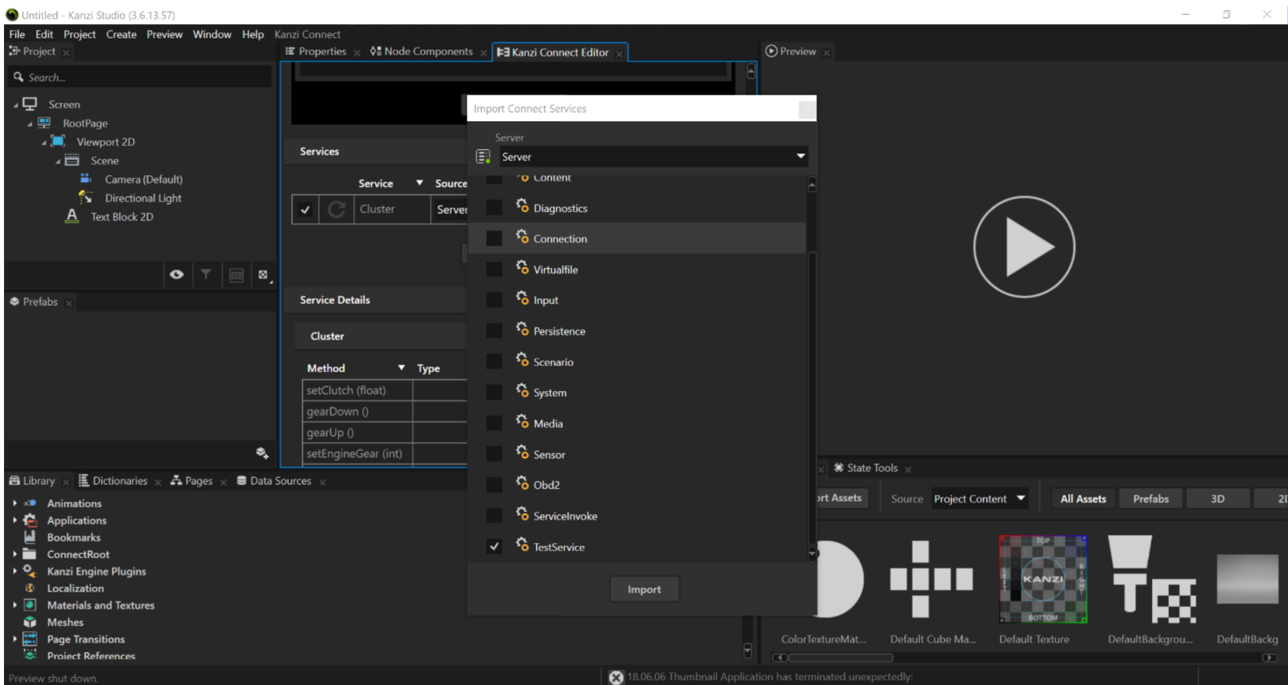

Figure 12. Importing a Kanzi Connect service to Kanzi Studio

Simulator can be used to simulate service functionality by for example invoking service methods and changing the state of service data. This way a Kanzi Connect service can be tested and prototyped without implementing it programmatically. To implement a simulated service its interface definition file can be exported from simulator and run through code generation scripts. The generated stub files can then be implemented with for example the C++ programming language.

4 Objectives

The objective of the thesis is to improve web-based communication with a Kanzi Connect server. Web-based communication with a Kanzi Connect server is currently limited to a stateless HTTP API. This chapter describes the specific problems that the thesis aims to solve.

4.1 State of Kanzi Connect Server

Simulator needs to know when the server is running and when it is not. It determines this by polling the server with an HTTP request every couple of seconds. This means that simulator does not have real-time knowledge of whether the server is running or not. The simulator user interface displays the online state of the server as shown in figure 13.

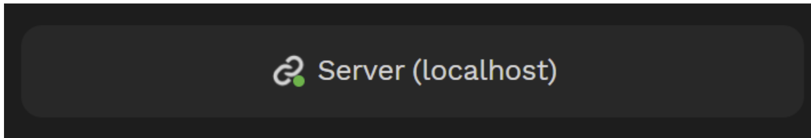

Figure 13. Server state indicator in Kanzi Connect Simulator

When simulator notices that the server changes state from offline to online it synchronizes state between itself and the server. The synchronization should only run when the server has actually changed state from offline to online because it interrupts user workflow with a loading spinner to prevent user modifications during the synchronization. Figure 14 shows the synchronization in action.

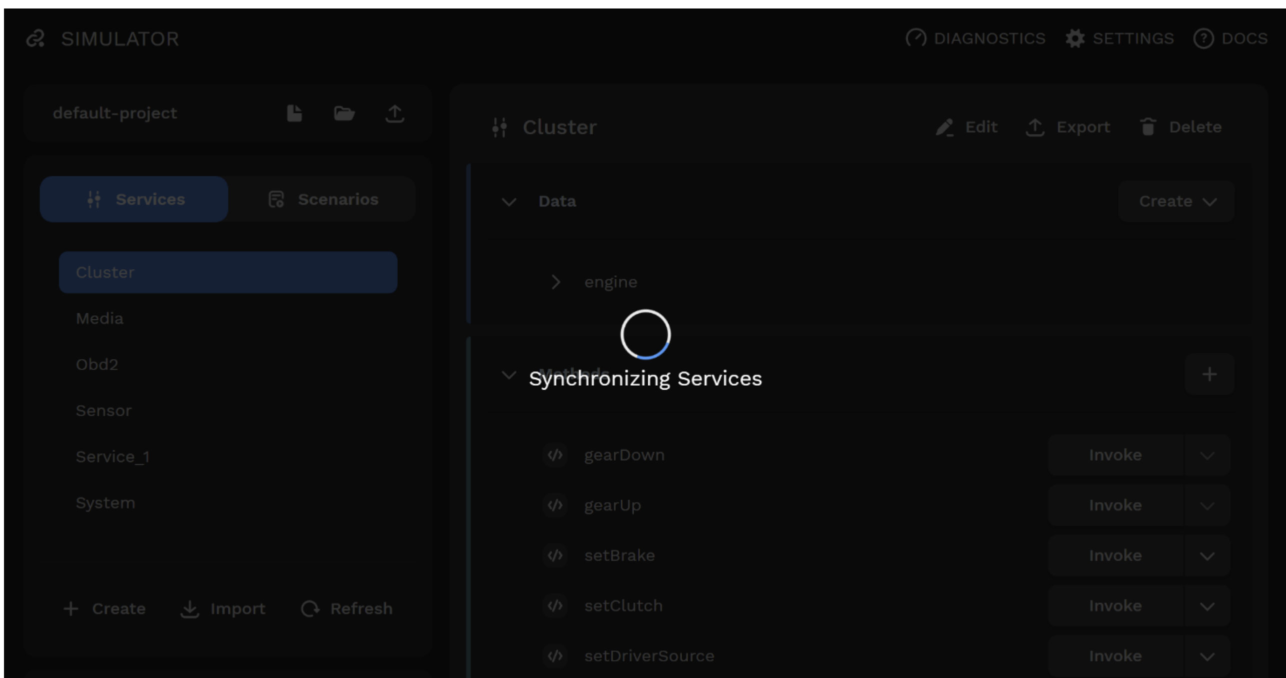

Figure 14. State synchronization in Kanzi Connect Simulator

Another reason why the synchronization should only run when necessary is that it can cause problems with different types of services. Services that load as part of server startup are real services with a C++ implementation. Services that the simulator creates on the server are always simulated without an implementation. For example, a real media service might have a method to play a track. A simulated service would have that same play method in its interface, but the method does not do anything. If during synchronization a service is both in the simulator and on the server, simulator tries to preserve user modifications. This means that if the user has modified the service in the simulator, simulator replaces the service on the server with the modified service. The side-effect of this is that simulator may replace a real service with a simulated service which can cause confusion to the user.

The first problem with determining the server state by polling is that simulator is used in a development environment where the server is often restarted through for example Visual Studio. If the server is restarted too quickly, simulator may not notice that the server restarted because it happened faster than the polling interval. As a result, the synchronization does not run and the services in the simulator do not reflect the services on the server.

The second problem is that the polling is unreliable and causes many false negatives. For example, changes in the network connection of a developer's computer can cause requests to timeout or otherwise fail and falsely indicate that the server is offline. Another situation where polling has problems is when the polling request gets stuck in a queue behind other HTTP requests and times out. This happens for example when a user rapidly changes the value of service data value with a slider controller. Figure 15 shows the slider controller in simulator. This causes a flood of HTTP requests that cause the polling request to timeout in a queue.

Figure 15. Slider controller in Kanzi Connect Simulator

Polling also adds flakiness to end-to-end tests. Simulator uses the Cypress end-to-end testing framework. Figure 16 shows the Cypress user interface. The tests sometimes fail because simulator cannot determine the state of the server and synchronize itself quickly enough.

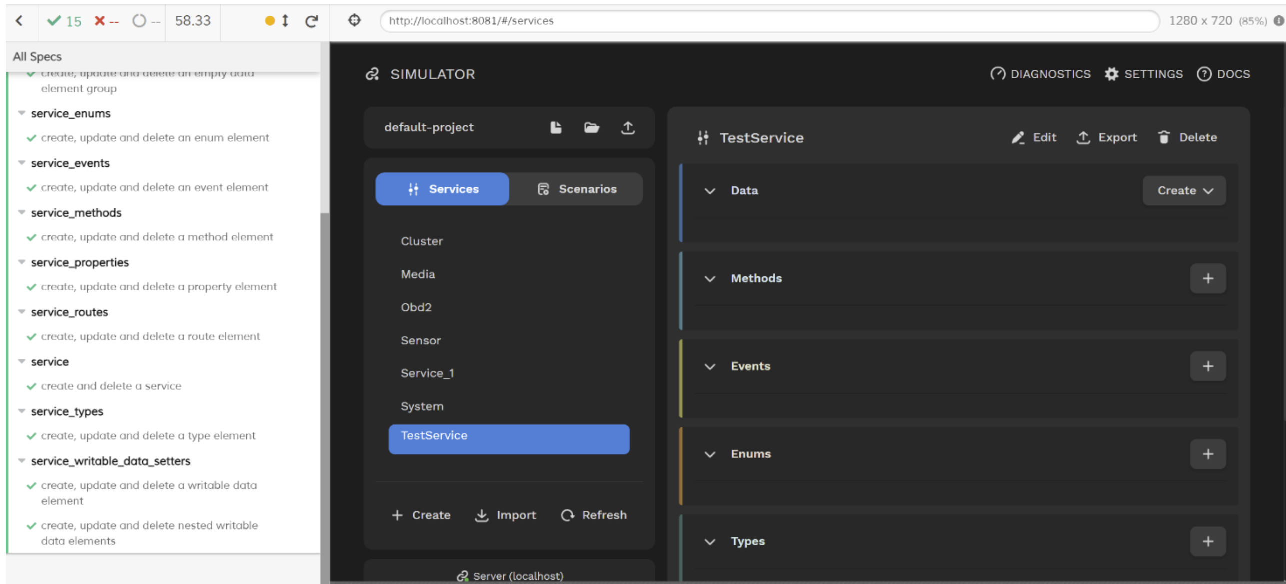

Figure 16. Cypress end-to-end test framework user interface

The final problem with polling is that it does not work when the network connection is slow. This has been observed when testing simulator on a mobile device through a USB connection to a laptop hosting the server. The polling requests take too long to complete and time out causing simulator to determine that the server is offline.

The objective is to find a reliable and real-time way of determining the state of the server. The assumption is that WebSocket would be suitable for this purpose.

4.2 Service Data Updates

Simulator can be used to change the value of service data elements using a variety of controllers. Figure 17 shows some of the available controllers such as an input field and a slider. Simulator updates the value to the server as an end user interacts with a controller. The problem is that while simulator can update service data values to the server, it cannot get real-time updates to service data values from the server. The value of a service data can change on the server through many

other means such as scenarios or a user interface developed with Kanzi Studio. The objective is to be able to get real-time updates to service data from the server using web technologies. The assumption is that WebSocket would enable this.

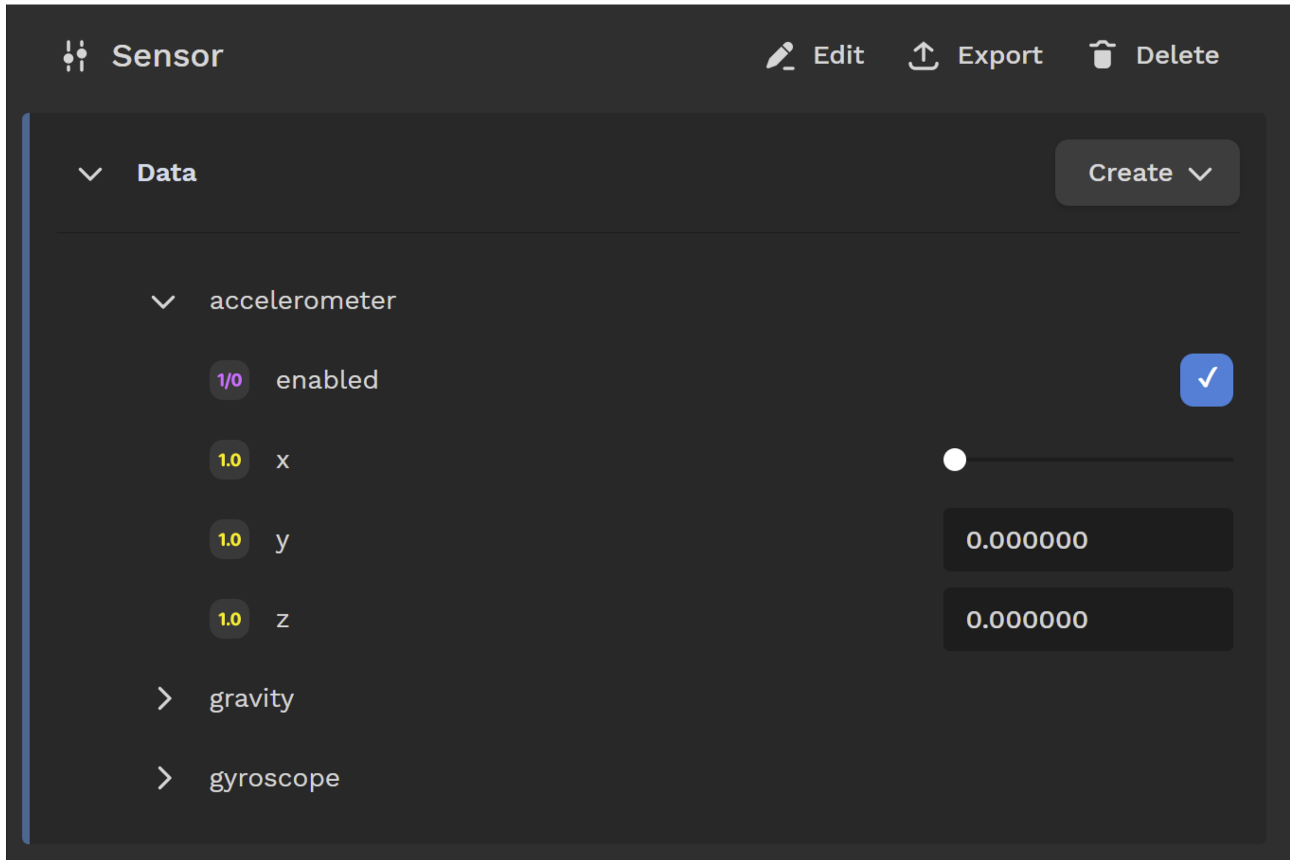

Figure 17. Service data controllers in Kanzi Connect Simulator

4.3 Service Event Notifications

A service can send events and applications developed with Kanzi Studio can listen to the events. Events are defined in the service interface. A developer can use simulator to modify and trigger service events as shown in figure 18. Triggering an event in the simulator instructs the service to send the event to clients. The problem is that the HTTP API does not provide a mechanism for getting notifications of triggered events. In other words, simulator can trigger events, but it cannot tell when a service event is triggered. The objective is to be able to listen to service events using web technologies. The assumption is that WebSocket would be suitable for this.

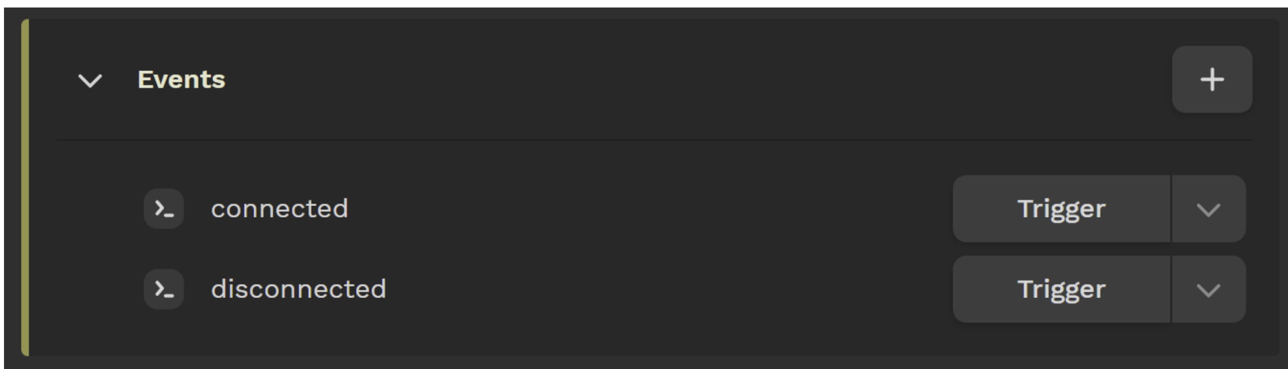

Figure 18. Service events in Kanzi Connect Simulator

4.4 Improving the HTTP API

Kanzi Connect has a stateless HTTP API as described in chapter three. The objective is to evaluate the HTTP API and its design to see whether it can be improved. While the HTTP API may have room for improvement, it is uncertain whether it can realistically be changed at this point of the lifecycle of the product. The assumption is that the API could be evaluated using the REST architectural style constraints.

5 Browser Networking

This chapter explores the APIs and protocols available to web applications in web browsers with the aim of finding a suitable technology for solving the problems described in chapter four. A web application can use a variety of application APIs and protocols to communicate with a server. No one protocol or API is the best in every situation. Nontrivial applications use a combination of transports based on requirements that may include areas such as reliability, caching, and latency. (Grigorik 2013, 258.)

5.1 HTTP

HTTP or Hypertext Transfer Protocol is an application-layer protocol that enables transmitting of hypermedia documents such as HTML. It was designed for communication between web browsers and web servers. HTTP uses a client-server model where a client opens a connection to make a request and then waits until it receives a response. HTTP is a stateless protocol which means that

the server does not keep any data or state between requests. HTTP is the foundation of data exchange on the Web. Due to its extensibility HTTP is also used to fetch content such as images and videos or to post content such as HTML form results to servers. HTTP can also be used to update web pages on demand by fetching parts of documents. (MDN Web Docs 2021b; MDN Web Docs 2021c.)

Requests and responses are the two types of HTTP messages. An HTTP request message consists of an HTTP method, a resource path, an HTTP protocol version, and optional headers. For some methods such as POST the message also has a body which contains the sent resource. (MDN Web Docs 2021c.) Code block 4 shows an example of an HTTP request message sent by Kanzi Connect Simulator. A resource path can optionally be followed by query parameters which are a list of key-value pairs separated with the ampersand symbol. The parameter list begins with a question mark symbol. A web server can implement extra functionality based on the parameters. (MDN Web Docs 2021f.)

```
GET /serviceinvoke/Media/method/play HTTP/1.1
Host: localhost:8080
Connection: keep-alive
Cache-Control: max-age=0
sec-ch-ua: " Not;A Brand";v="99", "Google Chrome";v="91", "Chromium";v="91"
sec-ch-ua-mobile: ?0
User-
Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Ge
cko) Chrome/91.0.4472.101 Safari/537.36
Accept: */*
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer: http://localhost:8080/app/index.html
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9,pl;q=0.8
```

Code Block 4. HTTP request sent by Kanzi Connect Simulator

An HTTP response message consists of an HTTP protocol version, an HTTP status code, a status message, and headers. The message can also optionally include a body containing the fetched resource. (MDN Web Docs 2021c.) Code block 5 shows an example of an HTTP response message received by Kanzi Connect Simulator from a Kanzi Connect server to the request in the previous code block.

```
HTTP/1.1 200 OK
Date: Sun, 18 Jul 2021 20:54:14 GMT
Content-Type: text/json
Content-Length: 39
Connection: close
Accept-Ranges: bytes

{
    "status": "OK",
    "value": ""
}
```

Code Block 5. HTTP response received by Kanzi Connect Simulator

HTTP defines a set of request methods or verbs that indicate the action that should be performed for a resource. Common HTTP request methods include GET, POST, PUT, PATCH, and DELETE. The GET method requests a representation of a resource and should only be used to retrieve data. The POST method submits an entity to the specified resource which often causes a change in state or other side effects on the server. The PUT method replaces representations of a resource with the request payload. The PATCH method applies partial modifications to a resource. The DELETE method deletes a resource. (MDN Web Docs 2021d.)

HTTP response status codes indicate whether an HTTP request completed successfully. Status codes are grouped in five classes:

- Informational responses (100 – 199)
- Successful responses (200 – 299)
- Redirects (300 – 399)
- Client errors (400 – 499)
- Server errors (500 – 599) (MDN Web Docs 2021e)

Informational class status codes (1xx) are interim responses that communicate connection status or request progress before a request is completed and a final response is sent. Successful class status codes (2xx) indicate that a request was successfully received, understood, and accepted. Redirection class status codes (3xx) indicate that the user agent must take further action for the request to be fulfilled. Client error class status codes (4xx) indicate that the client may have made a mistake.

Server error class status codes (5xx) indicate that the server may have made a mistake or that it cannot perform the requested method. (Fielding & Reschke 2014.)

5.2 XHR

XHR or XMLHttpRequest is an API provided by web browsers that is used to send HTTP requests in order to exchange data between a website and a server. XHR allows data to be retrieved from a URL without refreshing the full web page. (MDN Web Docs 2021g; MDN Web Docs 2021h.) Code block 6 shows an example XMLHttpRequest that retrieves the service description of a media service from a Kanzi Connect server.

```
function onLoad () {
  console.log(this.responseText)
}

let xhr = new XMLHttpRequest()
xhr.addEventListener("load", onload)
xhr.open("GET", "http://localhost:8080/servicedescriptions/media")
xhr.send()
```

Code Block 6. XMLHttpRequest example

XHR is not suitable for real-time notifications and requires the client to poll the server for updates. Polling is simple to implement but inefficient. Short polling intervals result in unnecessary traffic and overhead, and long polling intervals result in delayed updates. Long-polling can improve this by keeping the connection open until the server has an update available. This way data gets sent to the client immediately when it becomes available. (Grigorik 2013, 274-276.)

XHR is a great choice for communication that follows the HTTP request-response cycle. Its limitations are primarily in streaming and real-time delivery of data. Technologies such as server-sent events and WebSocket are simpler and more efficient options for these use cases. (Grigorik 2013, 278.)

5.3 Fetch

The Fetch API provides an interface for fetching resources. Fetch is essentially a modern replacement for XHR, and its API provides a more powerful and flexible feature set. Fetch provides generic definitions for network request related concepts including a Request and a Response object. These objects are usable anywhere where requests and responses need to be handled such as service workers and the Cache API. The fetch method takes a resource path as its only required argument and returns a Promise that resolves to a Response object. A number of methods exist to define and handle the body content of a Response object. (MDN Web Docs 2021a; MDN Web Docs 2021i.) Kanzi Connect Simulator currently uses Fetch for all communication with a Kanzi Connect server. Code block 7 uses Fetch to implement the same functionality as the XHR example shown in code block 6.

```
fetch("http://localhost:8080/servicedescriptions/media")
    .then(response => response.text())
    .then(text => console.log(text))
```

Code Block 7. Fetch example

5.4 SSE

SSE or Server-Sent Events enables server-to-client streaming of text-based data. SSE has two components: the EventSource API in the browser and the event stream data format used to deliver updates. SSE delivers messages over a single long-lived HTTP connection. The primary limitation of SSE is that it is unidirectional and does not allow the client to send messages to the server. (Grigorik 2013, 279, 285.)

The client-side code for working with SSE is similar to WebSocket in terms of handling incoming events. Creating an EventSource object with the URL of the SSE event stream resource opens a connection to the server and begins receiving events from it. A message handler can be used to listen to messages that do not have an event field on them. Specific events can be listened for by adding event listeners to the event source. (MDN Web Docs 2021j.) Code block 8 shows an example of these concepts.

```
const eventSource = new EventSource("http://localhost:3000/sse")

eventSource.onmessage = function(message) {
    console.log(`message: ${message.data}`)
}

eventSource.addEventListener("my-event", function(event) {
    console.log(`my-event: ${event.data}`)
})
```

Code Block 8. Server-sent events client-side example

The server-side code needs to respond with the text/event-stream MIME type. Messages are terminated by a pair of newlines. (MDN Web Docs 2021j.) Code block 9 shows a server-side example of SSE using Node.js and Express. The code registers an endpoint that a client can connect to and sends an event to the client each second.

```
const express = require('express')
const app = express()

const headers = {
    'Content-Type': 'text/event-stream',
    'Connection': 'keep-alive',
    'Cache-Control': 'no-cache',
    'Access-Control-Allow-Origin': '*'
}

app.get('/sse', function (req, res) {
    res.writeHead(200, headers)
    const intervalId = setInterval(() => {
        res.write('event: my-event\n')
        res.write('data: data\n\n')
    }, 1000)

    req.on('close', () => clearInterval(intervalId))
})

app.listen(3000, () => console.log("Listening on 3000"))
```

Code Block 9. Server-sent events server-side example

5.5 WebSocket

WebSocket allows a client and a server to communicate bidirectionally using text and binary data. In comparison to raw network sockets the WebSocket API provides many additional features such as connection negotiation, interoperability with existing HTTP infrastructure, and message-oriented communication. WebSocket is versatile and flexible with a simple and minimal API that allows layering of arbitrary application protocols between a client and a server. The trade-off with custom protocols is that the application must handle areas such as compression, caching, and state management that the browser would otherwise provide. (Grigorik 2013, 287.)

A client needs to create a WebSocket object to communicate with a server using the WebSocket protocol. This automatically attempts to open a connection to the server. The WebSocket constructor takes as arguments the URL to which to connect and an optional array of protocol strings. (MDN Web Docs 2021k.) Code block 10 shows an example of client-side WebSocket code.

```
const ws = new WebSocket('ws://localhost:8080/events')

ws.onopen = function () {
    console.log("connection opened")
}

ws.onclose = function () {
    console.log("connection closed")
}

ws.onmessage = function (message) {
    console.log(message.data)
}

ws.send("Hello from the WebSocket client!")
```

Code Block 10. WebSocket client-side example

A WebSocket server is an application that listens on any port of a TCP server that follows a specific protocol. Any server-side programming language that is capable of Berkeley sockets can be used to write a WebSocket server. (MDN Web Docs 2021l.) The CivetWeb web server embedded in a Kanzi Connect server includes support for server-side WebSocket functionality (CivetWeb 2021).

5.6 WebRTC

WebRTC or Web Real-Time Communication is a group of protocols, standards, and JavaScript APIs that together allow peer-to-peer audio, video, and data sharing between browsers. WebRTC makes real-time communication possible without the use third-party plugins or proprietary software. WebRTC enables use cases such as audio and video teleconferencing in the browser. (Grigorik 2013, 309.) It can also be used for simpler web applications that use the camera or microphone. The technology can be used in all modern browsers as well as on native clients of major platforms. WebRTC is available in browsers as JavaScript APIs. Native clients such as Android and iOS applications use a library that provides the same functionality. (WebRTC n.d.)

6 Web APIs

This chapter explores Web APIs and API design with the goal of forming a basis from which to evaluate the Kanzi Connect HTTP API. The focus is on request-response APIs. There are also event-driven APIs which can be implemented with technologies such as Webhooks, WebSocket, or SSE which is a form of HTTP Streaming (Jin et al. 2018; Stack Overflow 2017). Web application programming interfaces or APIs are critical to the modern world, and they are everywhere. At its core an API is a point where two systems or other entities meet and interact. An entity that uses an API is called a consumer, and an entity that exposes an API is called a provider. There are different types of APIs such as system APIs, libraries, and remote APIs. Web APIs are remote APIs that use the HTTP protocol. (Lauret 2019, 3-6.)

6.1 API Design

API design is the foundation of APIs, and the success or failure of an API-based system directly depends on the quality of its API design. API design is important because APIs are used by people who expect the interfaces to be helpful and simple. This is no different from any other interface such as a website or an everyday object. A poorly designed interface is frustrating to use and can even be dangerous. People are unlikely to want to use a poorly designed interface again. A flawed API can be misused, underused, or not used at all. Building software using a flawed API takes more time, effort, and money. Users of a flawed API need more support from the API provider further increasing costs. Flawed API design can also result in security vulnerabilities. These are just some of the harmful effects of poorly designed APIs. (Lauret 2019, 9-14.)

The goal of an API is to allow developers to reach their goal as simply as possible regardless of the technology. Technologies such as RPC, REST, and GraphQL all enable software communication over a network. These can be considered as API styles. While all API styles have their own common practices, fundamental principles of API design apply to any API style. (Lauret 2019, 14-15.)

The cornerstone of API design is to determine the goals that a consumer of the API wants to achieve. An API should be designed from the perspective of what users can do, not from the perspective of how the software operates. In other words, an API should be designed from its consumer's perspective and not its provider's. An API should not expose implementation details. Users of an API want to achieve their goals and they do not care about exactly how the API functions internally. An API designed from the provider's perspective will be complicated to use and presents goals that only make sense for the provider. (Lauret 2019, 17-24.)

When designing an API, it is important to have accurate knowledge of who can use the API, what they can do, how they do it, what they need to do it, and what they get in return. What users can do and how do they do it are fundamental questions to ask. The answers to the "what" question are decomposed into how questions, with each how question corresponding to a goal. For example, users can buy products and they do it by adding products to a cart and checking out. Adding products to a cart and checking out the cart become the goals. A goal might also take inputs and return outputs. For example, checking out a cart requires a shopping cart as an input, and it returns an order confirmation as an output. To identify any missing goals, it is also helpful to ask where the inputs come from and how the outputs are used. (Lauret 2019, 24-33.)

An API should not expose internal business logic as this can make the API hard to use and understand for the consumer and dangerous for the provider. A consumer might use the API incorrectly and compromise data integrity. Exposing the software architecture in the API causes similar issues. For example, a system that stores product descriptions and pricing in different systems might mirror the architecture by exposing separate endpoints for retrieving the description and price of a product. A better approach would be to expose a single endpoint for searching products and let the API implementation gather the necessary information. Exposing the human organization in the API should also be avoided. For example, preparing and shipping an order are internal processes that

are irrelevant to a consumer of the API. Figure 19 shows an API goals canvas with questions that can be asked to determine an API's goals. (Lauret 2019, 38-41.)

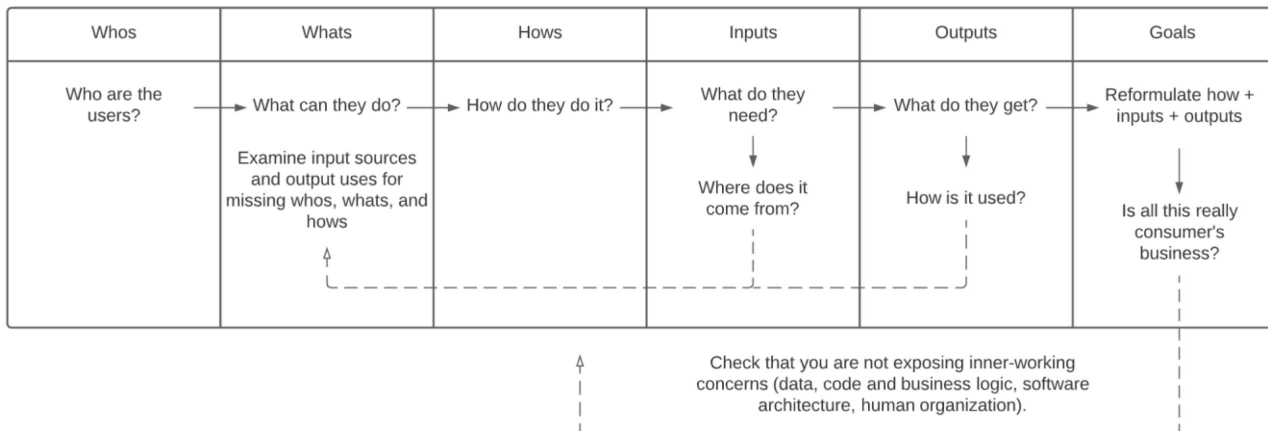

Figure 19. API goals canvas

A well-designed web API should support platform independence. The API should be usable by any client regardless of how the API is internally implemented. This is achieved by using standard protocols and by having a mechanism for the client and the web service to agree on the data exchange format. A web API should also support service evolution and be able to evolve and add functionality independently from client applications. Existing client applications should continue to function without modifications when the API evolves. (Microsoft Docs 2018.) A well-designed API is easy to read and work with, hard to misuse, and complete and concise (Swagger 2021b).

6.2 RPC APIs

RPC or Remote Procedure Call is a simple API paradigm where a client executes a block of code on a server. RPC is about actions, whereas REST is about resources. Clients send a method name and arguments to a server which responds in JSON or XML. In RPC APIs the endpoints contain the name of the executed operation. The most fitting HTTP method is used, which is typically HTTP GET method for read-only requests and POST for everything else. RPC works well for APIs with a variety of actions that do not match CRUD operations (Create, Read, Update, Delete) or that have side effects unrelated to the target resource. (Jin et al. 2018, 13.)

RPC-based APIs are great for actions, whereas REST-based APIs are great for modeling a domain and having CRUD operations available for all the data (Sturgeon 2016). The RPC model is an inverse of the REST model. In RPC the addressable units are procedures, and the problem domain entities are hidden behind the procedures. In REST the addressable units are the entities, and behaviors of the system are hidden behind the entities as side-effects of creating, updating, or deleting them. (Nally 2018.)

The Web API for the Slack communication platform is one example of an RPC-style API. The Slack Web API is a collection of HTTP RPC-style methods that follow the form shown in code block 11. (Slack API 2021.)

```
// Form of Slack Web API URLs.
https://slack.com/api/METHOD_FAMILY.method

// Slack Web API example with a JSON-encoded body.
POST /api/conversations.create
Content-type: application/json
Authorization: Bearer xoxp-xxxxxxxxx-xxxx
{"name":"something-urgent"}
```

Code Block 11. Slack RPC Web API

6.3 REST APIs

A REST API, or RESTful API, is an API that conforms to the REST architectural style constraints. REST stands for Representational State Transfer and it is an architectural style introduced by Fielding (2000). A REST API allows consumers to manipulate resources identified by paths using standardized HTTP methods. For example, in the HTTP request GET /products/{productId}, the path /products/{productId} identifies a product resource and the GET HTTP method represents the retrieve action applied to the resource. (Lauret 2019, 44, 48, 72-74.) REST is often misunderstood, and many HTTP-based APIs are incorrectly labeled as RESTful (Fielding 2008; Sturgeon 2017).

For a software architecture to be considered RESTful it needs to conform to the following six constraints:

- Client-server separation
- Statelessness
- Uniform interface
- Cacheability
- Layered system
- Code on demand (Lauret 2019, 73-74)

Client-server separation refers to separation of concerns. Separating the client from the server improves portability of the client across platforms and improves scalability by simplifying the server components. Client-server separation allows the components to evolve independently. (Fielding 2000, 78.) The client application should only know the URI of the requested resource and it should not interact with the server application in any other way. The server application similarly should not modify the client application in other ways than passing it to the requested data via HTTP. (IBM Cloud Education 2021.)

Statelessness means that communication must be stateless in nature. Each request from a client to a server must contain all necessary information to understand the request. Requests cannot take advantage of stored context on the server. (Fielding 2000, 78-79.) In other words, a REST API does not require server-side sessions (IBM Cloud Education 2021). Each request is separate and unconnected (Red Hat 2020). This improves visibility because a monitoring system can determine the full nature of the request based on a single request. Reliability is improved because it becomes easier to recover from partial failures. Scalability is improved because the server can quickly free resources as it does not need to store state between requests. Because the server does not need to manage resource usage across requests the server implementation is also simplified. (Fielding 2000, 78-79.)

The emphasis on a uniform interface between components is the central feature that distinguishes the REST architectural style from other network-based styles. Generality in the component interface simplifies system architecture and improves the visibility of interactions. Independent evolvability is encouraged by decoupling implementations from the services they provide. The trade-off is that efficiency is downgraded as information is transferred in a standardized form instead of one that is specific to an application's needs. Four interface constraints guide the behavior of components in order to obtain a uniform interface: identification of resources, manipulation of resources through representations, self-descriptive messages, and hypermedia as the engine of application state. (Fielding 2000, 81-82.)

Cacheability means that the data within a response to a request must be implicitly or explicitly marked as cacheable or non-cacheable. A client cache has the right to reuse response data for later equivalent requests if the response is cacheable. The cache constraint has the potential to eliminate some interactions partially or completely, which improves efficiency, scalability, and user-perceived performance. The downside is that a cache can decrease reliability if stale data within the cache differs significantly from the data that would have been obtained from the server. (Fielding 2000, 79-80.)

The layered system constraint enables an architecture to be composed of hierarchical layers by allowing each component to only see the immediate layer with which it is interacting (Fielding 2000, 82-83). Calls and responses go through different layers in a REST API, and the client and the server may not be directly connected to each other. The communication loop may have a number of different intermediaries. A REST API should be designed so that neither the client nor the server knows whether it is communicating with the end application or an intermediary. (IBM Cloud Education 2021.) Intermediaries may be responsible for tasks such as load-balancing or security (Red Hat 2020).

Code on demand means that REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This improves system extendibility and simplifies clients by reducing the required number of pre-implemented features. Code on demand is an optional constraint within REST because it also reduces visibility. (Fielding 2000, 84.)

A well-known model for assessing the compliance of RESTful API implementation is the Richardson maturity model. The model has four maturity levels of implementation from zero to three. Figure 20 depicts the model. (Santoro et al. 2019, 10.) Level three of the Richardson maturity model is a pre-condition of REST (Fielding 2008).

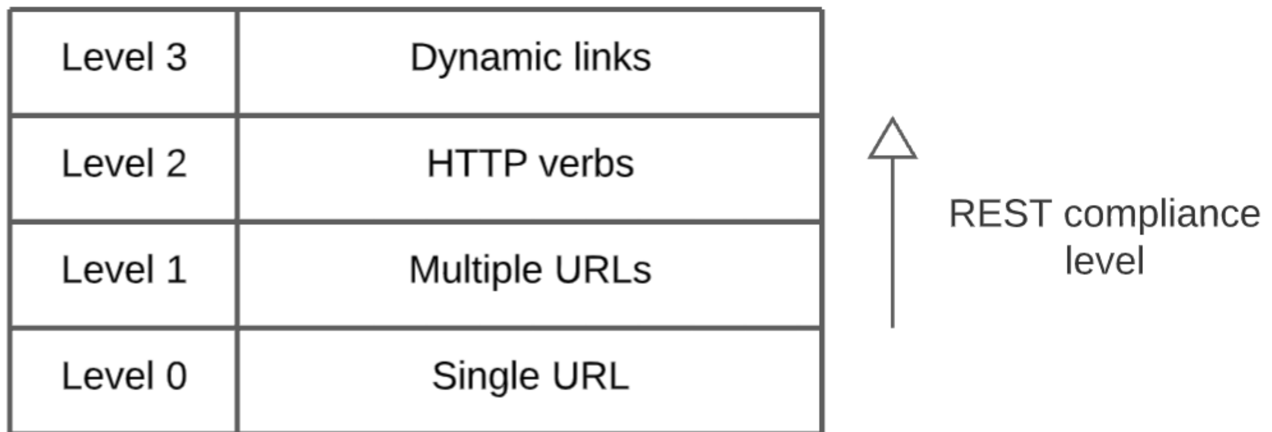

Figure 20. Richardson maturity model

Level zero services have a single URI and use a single HTTP method (Webber et al. 2010, 19). HTTP is used as a transport system but without any of the mechanisms of the web (Fowler 2010). APIs at this level are essentially RPC APIs over the network topology built around the HTTP protocol (Santoro et al. 2019, 11).

Level one services use multiple URIs but only a single HTTP verb. While level zero services tunnel all interactions through a single resource, level one services expose multiple logical resources. Operation names and parameters are inserted into a URI which is transmitted to a remote service. Most services that describe themselves as RESTful today are often level one services. Although they do not adhere to RESTful constraints, level one services can be useful. It can however be possible for example to accidentally destroy data using the GET verb which should not have such side effects. (Webber et al. 2010, 19-20.)

Level two services support several of the HTTP verbs for each exposed resource. This level includes CRUD services. Level two services use HTTP verbs and status codes to coordinate interactions, which means that they use the Web for robustness. (Webber et al. 2010, 20.)

The final level introduces HATEOAS or Hypertext As The Engine Of Application State (Fowler 2010). Level three services include URI links to other resources that might of interest to consumers in representations. Consumers are lead through a trail of resources that results in application state transitions. (Webber et al. 2010, 20.) Hypermedia controls in the response describe what can be done

next, and the URI of the resource needed to do it. For example, getting a list of appointments might include a URI that describes how to book an appointment. (Fowler 2010.)

GitHub is one example of a platform that has a REST API (GitHub Docs 2021a). Code block 12 shows an example of a request to the GitHub REST API and its response.

```
// Request.
GET https://api.github.com/users/lmalmber

// Response.
{
    "login": "lmalmber",
    "id": 6433355,
    "node_id": "MDQ6VXNlcjY0MzMzNTU=",
    "avatar_url": "https://avatars.githubusercontent.com/u/6433355?v=4",
    "gravatar_id": "",
    "url": "https://api.github.com/users/lmalmber",
    "html_url": "https://github.com/lmalmber",
    "followers_url": "https://api.github.com/users/lmalmber/followers",
    "gists_url": "https://api.github.com/users/lmalmber/gists{/gist_id}",
    "starred_url": "https://api.github.com/users/lmalmber/starred{/owner}{/repo}",
    "subscriptions_url": "https://api.github.com/users/lmalmber/subscriptions",
    "organizations_url": "https://api.github.com/users/lmalmber/orgs",
    "repos_url": "https://api.github.com/users/lmalmber/repos",
    "events_url": "https://api.github.com/users/lmalmber/events{/privacy}",
    "received_events_url": "https://api.github.com/users/lmalmber/received_events",
    "type": "User",
    "site_admin": false,
    "name": "Lasse Malmberg",
    ...
}
```

Code Block 12. GitHub REST API request and response example

6.4 GraphQL APIs

GraphQL is a query language for APIs. It allows clients to define the structure of the required data, and the server returns that structure. GraphQL APIs need only a single URL endpoint. GraphQL APIs do not need different HTTP methods to describe the operation. The JSON body of the request indicates whether the request is a query or a mutation. (Jin et al. 2018, 15.)

GraphQL has a few advantages over REST and RPC. GraphQL saves multiple round trips by enabling clients to nest queries and fetch data across resources with a single request. This might otherwise require multiple HTTP calls to the server. GraphQL avoids versioning by allowing the API to be extended with new fields and types without affecting existing queries. REST and RPC API responses can include data not needed by the client. With GraphQL payload sizes can be smaller because clients specify exactly what they need. GraphQL is also strongly typed which means that type checking can be used at development time to ensure that a query is valid. One of the drawbacks of GraphQL is that it adds complexity for the API provider. The server needs to do more processing in the form of parsing complex queries and verifying parameters. (Jin et al. 2018, 16-17.)

In addition to a REST API GitHub also has a GraphQL API. Code block 13 shows an example GraphQL request to the API and its response. Communication with the GitHub GraphQL API requires an OAuth token with the right scopes. (GitHub Docs 2021b; GitHub Docs 2021c.)

```
// Request.
POST https://api.github.com/graphql
{
    user(login:"lmalmber") {
        name
        url
        organization(login:"rightware") {
            name
        }
    }
}

// Response.
{
    "data": {
        "user": {
            "name": "Lasse Malmberg",
            "url": "https://github.com/lmalmber",
            "organization": {
                "name": "Rightware"
            }
        }
    }
}
```

Code Block 13. GitHub GraphQL API request and response example

6.5 OpenAPI

OpenAPI or the OpenAPI specification (OAS) defines a standard, language-agnostic interface to RESTful APIs. It allows the capabilities of a service to be discovered and understood by both humans and computers without access to the source code or documentation. When properly defined, a consumer can understand and interact with a remote service with minimal implementation logic. An OpenAPI definition enables many use cases. For example, documentation generation tools can use an OpenAPI definition to display the API, and code generation tools can generate servers and clients in various programming languages. (Swagger 2021a.) Code block 14 shows an example OpenAPI definition written in YAML.

```yaml
openapi: 3.0.0
info:
  title: OpenAPI Example
  description: This is an example OpenAPI definition.
  version: 1.0.0
servers:
  - url: http://localhost:8080/
    description: Kanzi Connect server on local machine
tags:
- name: "service"
paths:
  /services:
    get:
      summary: Returns a list of services.
      tags:
      - "service"
      responses:
        '200':
          description: A JSON array of services
          content:
            application/json:
              schema:
                type: array
                items:
                  type: object
```

Code Block 14. OpenAPI definition example

OpenAPI definition files can be edited and previewed with for example the Visual Studio Code editor using the OpenAPI Editor and OpenAPI Preview extensions. Figure 21 shows the documentation preview in Visual Studio Code for the OpenAPI definition example shown in code block 14.

OpenAPI Example 1.0.0 OAS3

This is an example OpenAPI definition.

Servers
http://localhost:8080/ - Kanzi Connect server on local machine

service

GET /services Returns a list of services.

Parameters
Try it out

No parameters

Responses

Code	Description	Links
200	A JSON array of services Media type application/json Controls Accept header. Example Value | Schema [{...}]	No links

Figure 21. OpenAPI definition preview in Visual Studio Code

7 Development

This chapter covers the development part of the thesis. The first half describes the development of a WebSocket API and its corresponding client-side functionality in Kanzi Connect Simulator. The second half evaluates the existing HTTP API.

7.1 WebSocket API

WebSocket was chosen as the technology for solving the problems described in chapter four. WebSocket is the most suitable option because it solves the existing problems while also providing the most flexibility for any future additions to the functionality.

7.1.1 Server-Side Implementation

The WebSocket API is implemented in the Kanzi Connect server as a plugin similar to certain other features such as policies and scripting support. The plugin is configurable in the server configuration file “connect_server_config.xml” as shown in figure 22.

```
<websocket>
  <attribute name="library" value="websocketplugin_default" />
  <attribute name="metaclass" value="connect.websocket.default" />
</websocket>
```

Figure 22. WebSocket plugin in Kanzi Connect server configuration file

The WebSocket plugin uses the CivetWeb web server embedded in Kanzi Connect server for WebSocket support. The plugin implements and registers WebSocket handler classes that inherit from the CivetWebSocketHandler class that the CivetWeb web server provides. Each WebSocket endpoint has its own handler class. The plugin consists of a RuntimedataWebSocketHandler class, an EventWebSocketHandler class, and a root WebSocket handler that is an empty default handler. Figure 23 shows the relationship of the handler classes.

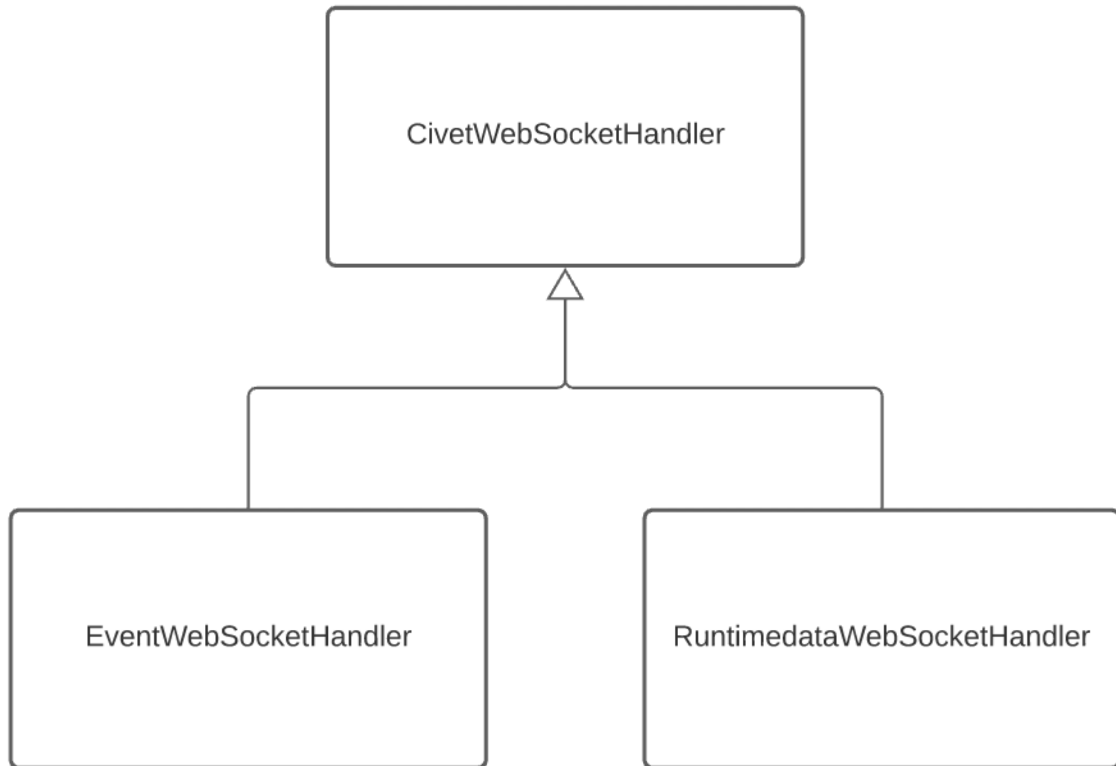

Figure 23. WebSocket plugin handler classes

The CivetWebSocketHandler class has four virtual functions that the plugin's WebSocket handler classes implement: handleConnection, handleReadyState, handleData, and handleClose. Figure 24 shows the interface for CivetWebSocketHandler. Function signatures are omitted. The handleConnection callback function is called when a client intends to open a WebSocket connection. The handleReadyState callback function is called when the WebSocket handshake successfully completes. The handleData callback function is called when data has been received from a client. The handleClose callback function is called when a client disconnects.

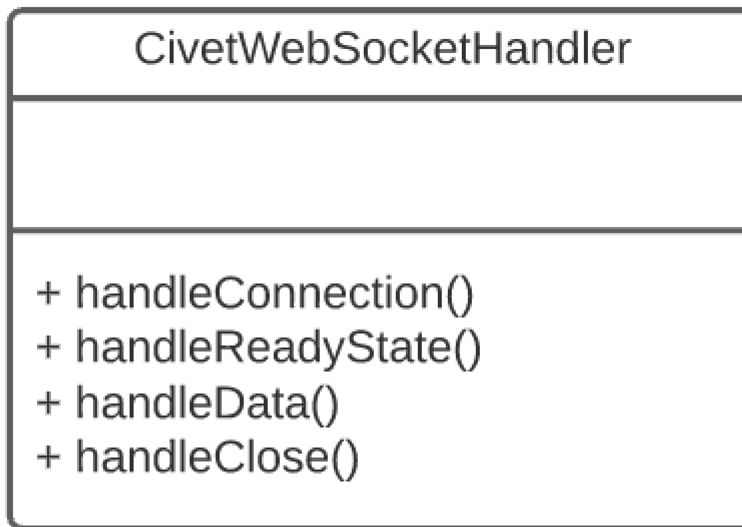

Figure 24. CivetWebSocketHandler class

The plugin first registers the root WebSocket handler which allows clients to connect to the "/" endpoint using WebSocket. This enables simulator to determine the state of the server using the WebSocket API. The root handler does not have any special functionality and it uses an instance of the CivetWebSocketHandler with default behavior. The plugin registers the handlers as part of its initialization as seen in code block 15.

```
void WebSocketContext::lateInitialize(ServiceManagerSharedPtr serviceManager)
{
    m_serviceManager = serviceManager;
    m_clientProxy = ConnectedClientProxy::create("websocket_client", m_domain.lock(),
make_shared<StubMessageDispatcher>(), "socket");
    m_runtimedataWebSocketHandler = std::make_unique<RuntimedataWebSocketHandler>();
    m_eventWebSocketHandler = std::make_unique<EventWebSocketHandler>(m_serviceManager,
m_clientProxy);

    if (auto httpServer = m_httpServer.lock())
    {
        httpServer->addWebSocketHandler(m_rootUri, new CivetWebSocketHandler());
        httpServer->addWebSocketHandler(RuntimedataWebSocketHandler::getUri(),
m_runtimedataWebSocketHandler.get());
        httpServer->addWebSocketHandler(EventWebSocketHandler::getUri(),
m_eventWebSocketHandler.get());
    }
}
```

Code Block 15. WebSocket handler registration

The RuntimedataWebSocketHandler class inherits from the CivetWebSocketHandler class. It allows clients to connect to the “/runtimedata” endpoint using WebSocket. Figure 25 shows the class interface.

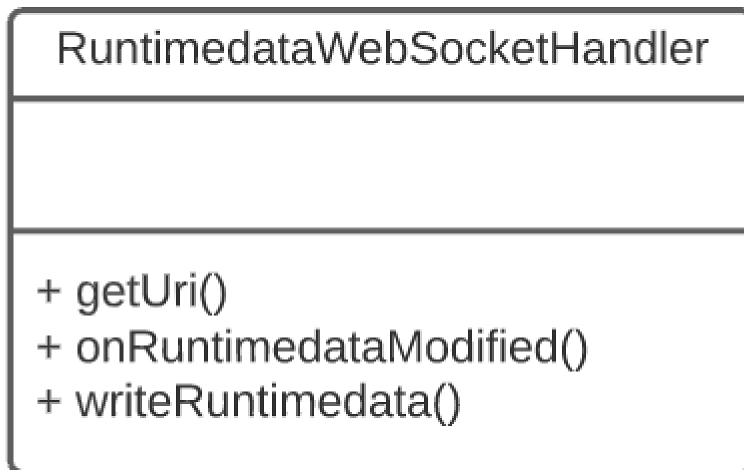

Figure 25. RuntimedataWebSocketHandler class

The class keeps track of client connections by storing them in a container. The handleReadyState function adds connections and the handleClose function removes connections from the container as seen in code block 16. This allows multiple clients to connect to the endpoint simultaneously.

```
void RuntimedataWebSocketHandler::handleReadyState(CivetServer* server, struct
mg_connection* conn)
{
    UNUSED_PARAMETER(server);
    lock_guard<mutex> lock(m_mutex);
    m_connections.push_back(conn);
}

void RuntimedataWebSocketHandler::handleClose(CivetServer* server, const struct
mg_connection* conn)
{
    UNUSED_PARAMETER(server);
    lock_guard<mutex> lock(m_mutex);
    auto connection = std::find(m_connections.begin(), m_connections.end(), conn);
    if (connection != m_connections.end())
    {
        m_connections.erase(connection);
    }
}
```

Code Block 16. Managing WebSocket handler connections

The WebSocket plugin is notified of changes to the values of Kanzi Connect RuntimeDataObjects. The plugin then instructs the RuntimedataWebSocketHandler to send a notification of the change in JSON format to all clients that are connected to the WebSocket endpoint as seen in code block 17.

```
void RuntimedataWebSocketHandler::writeRuntimedata(RuntimeDataObject* runtimedataObject)
{
    // Parse service name and path from full path. For example, "Cluster" and "engine.speed"
from "Cluster.engine.speed".
    kanzi::string fullPath = runtimedataObject->getPath();
    kanzi::string serviceName = fullPath.substr(0, fullPath.find('.'));
    kanzi::string path = fullPath.substr(kanzi::min(fullPath.length(), serviceName.length()
+ 1), fullPath.length());

    // Format the information as a JSON message.
    kanzi::string jsonMessage = "{";
    jsonMessage += "\"service\": \"" + serviceName + "\", ";
    jsonMessage += "\"path\": \"" + path + "\", ";
    jsonMessage += "\"value\": \"" + runtimedataObject->getSerializedValue() + "\"";
    jsonMessage += "}";

    for (const auto& connection : m_connections)
    {
        mg_lock_connection(connection);
        mg_websocket_write(connection, MG_WEBSOCKET_OPCODE_TEXT, jsonMessage.c_str(),
strlen(jsonMessage.c_str()));
        mg_unlock_connection(connection);
    }
}
```

Code Block 17. Sending runtime data changes to clients

Figure 26 shows an example of a message sent by the RuntimedataWebSocketHandler. This allows Kanzi Connect Simulator to listen to server-side changes to service runtime data and update its user interface in real-time to match the state of the server.


```json
{
  "service": "System",
  "path": "demo.charge",
  "value": "100"
}
```


Figure 26. JSON message sent by RuntimedataWebSocketHandler

The EventWebSocketHandler class also inherits from the CivetWebSocketHandler class. It allows clients to connect to the "/events" endpoint using WebSocket. Figure 27 shows the class interface.

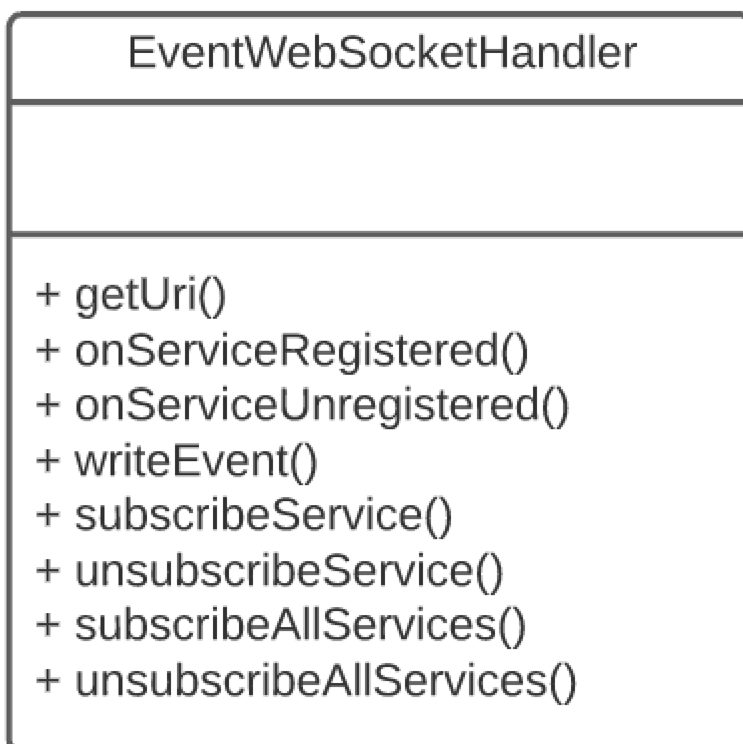

Figure 27. EventWebSocketHandler class

EventWebSocketHandler keeps track of client connections similar to RuntimedataWebSocketHandler. The EventWebSocketHandler class subscribes to all service events using a server-side client proxy and message subscriptions as seen in code block 18. It uses a regular expression message subscription to match all messages that are events. The ability to use regular expressions as message subscriptions had to be added as a new feature.

```
optional<int> sessionId = m_clientProxy->acquireSession(service);
if (sessionId)
{
    m_serviceSessions[service] = sessionId.value();
    auto session = m_clientProxy->getSession(*sessionId);
    auto callback = [this](const MessagePackage& message) -> bool { this-
>writeEvent(message); return false; };
    session->addRegexMessageSubscription("(.*)Event", callback);
}
```

Code Block 18. Subscribing to service events

The plugin is notified of service register and unregister events. The EventWebSocketHandler uses this information to register and unregister to events of services dynamically as services are added or removed. As EventWebSocketHandler receives event messages from the message subscriptions, it processes the messages and sends events to all connected clients in JSON format as seen in code block 19.

```
void EventWebSocketHandler::writeEvent(const MessagePackage& message)
{
    const kanzi::string serviceName = message.getInterfaceIdentifier();

    // Parse plain event name from the message type. For example, "progress" from
"MediaProgressEvent".
    kanzi::string eventName = message.getType();
    eventName.erase(0, serviceName.size());
    eventName.erase(eventName.rfind("Event"));
    eventName[0] = tolower(eventName[0]);

    // Parse event arguments from the message as strings.
    kanzi::vector<kanzi::string> arguments;
    for (int i = MessagePackage::ATTRIBUTE_KEY_ARGUMENT_1; i <=
MessagePackage::ATTRIBUTE_KEY_ARGUMENT_101; ++i)
    {
        // Omitted.
    }

    // Format the event as a JSON message.
    kanzi::string jsonMessage = "{";
    jsonMessage += "\"service\": \"" + serviceName + "\", ";
    jsonMessage += "\"event\": \"" + eventName + "\", ";
    jsonMessage += "\"args\": [";
    for (size_t i = 0; i < arguments.size(); ++i)
    {
       // Omitted.
    }
    jsonMessage += "]}";

    for (const auto& connection : m_connections)
    {
        mg_lock_connection(connection);
        mg_websocket_write(connection, MG_WEBSOCKET_OPCODE_TEXT, jsonMessage.c_str(),
strlen(jsonMessage.c_str()));
        mg_unlock_connection(connection);
    }
}
```

Code Block 19. Sending events to clients

Figure 28 shows an example of a message sent by the EventWebSocketHandler.


```
{
  "service": "Service",
  "event": "serviceRegistered",
  "args": ["Media"]
}
```


Figure 28. JSON message sent by EventWebSocketHandler

7.1.2 Client-Side Implementation

Kanzi Connect Simulator keeps a WebSocket connection open to each of the three WebSocket end-points exposed by the WebSocket API: “/”, “/runtimedata”, and “/events”.

The root endpoint “/” is used to determine whether the server is online. The WebSocket connection to the endpoint closes immediately if the server goes offline. When the server is offline simulator attempts to reconnect to the root endpoint by polling the server. Simulator only attempts to connect to the other two endpoints once it connects to the root endpoint and determines that the server is online. Otherwise, the existing functionality associated with the server state is the same as before. For example, simulator synchronizes state with the server once the connection to the root WebSocket endpoint succeeds.

Simulator uses the “/runtimedata” endpoint to listen to changes to service runtimedata values. The server sends a JSON message for each runtimedata value modification. Simulator finds a service runtimedata object in its own state based on the path value in the JSON message. It then sets the value of the runtimedata object to the value in the JSON message as seen in code block 20.

```
runtimeDataWebSocket.onmessage = async message => {
    try {
        message = JSON.parse(message.data)
    }
    catch {
        // Some runtime data such as Scenario service scripts fail JSON parsing.
        return
    }

    const service = rootState.services.services.find(service => service.name == mes
sage.service)
    if (service) {
        const path = message.path.split(".")
        let dataElement = service.data.findElementByPath(path)
        if (dataElement) {
            dataElement.setValueFromWebSocket(message.value)
        }
    }
}
```

Code Block 20. Handling WebSocket runtime data updates

Using this functionality simulator is able to display the state of service runtimedata objects in its user interface in real-time as the values update on the server. For example, in the case of the Cluster service the values seen in figure 29 update tens of times per second as the server-side simulation calculates new values for the runtimedata objects.

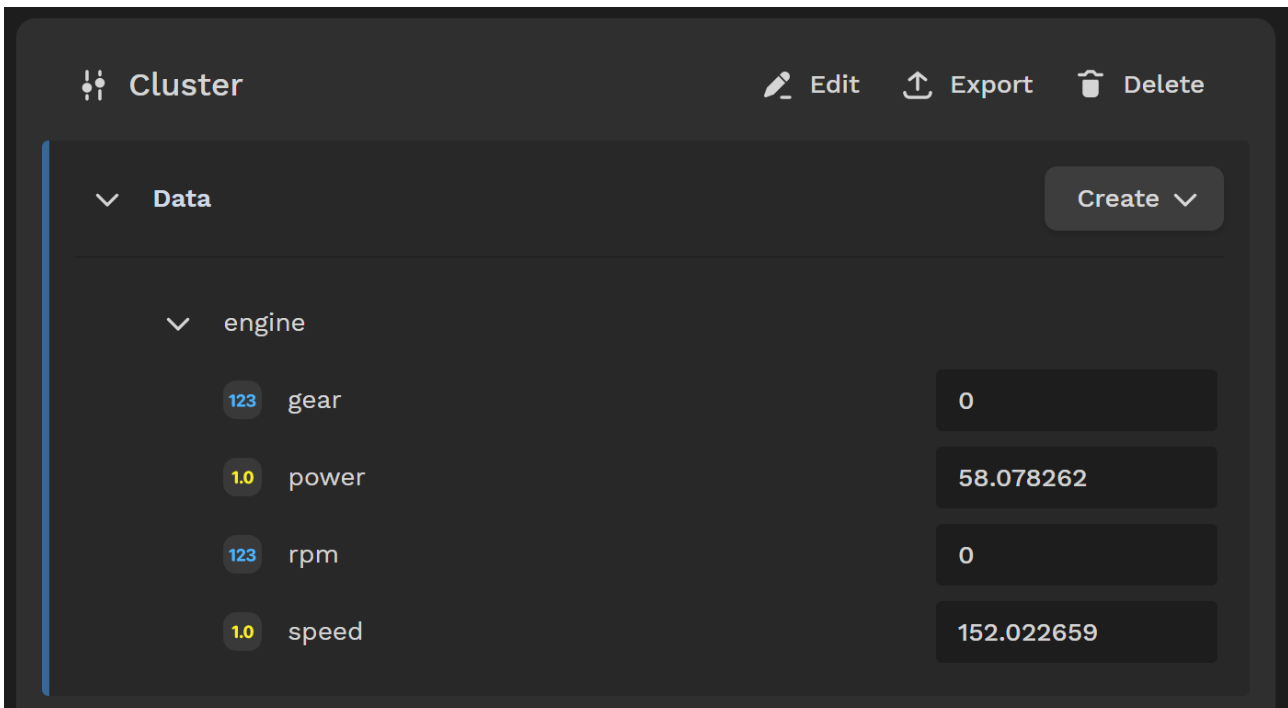

Figure 29. Cluster service runtime data objects in Kanzi Connect Simulator

The exception handling in the code is required because certain runtime data values cause problems with the JSON parsing. Due to how for example the scenario service is implemented server-side, it sends updates of scenario JavaScript scripts to the runtime data WebSocket endpoint which are not valid JSON.

Another observation is the "setValueFromWebSocket" function for setting the value of a data element from the WebSocket handler. This is required to prevent a problem with self-updates. As the value of a data element is changed using the simulator user interface, the updates immediately come back to simulator from the server through WebSocket and set the value of the data element to the same value again. This produces a strange effect where for example the value displayed in a text input jumps back and forth as an end user types a value. For example, typing "123456" in a text field might end up as "1256". The reason for this is that there is no way to identify who caused a

change to the value of a data element on the server. As a result, simulator gets updates from WebSocket to changes that it caused itself, but it is unable to recognize and filter these updates out. The solution is that simulator locks a data element from WebSocket updates for one second whenever the data element's value is modified through the user interface. This lock is used in the setValueFromWebSocket function to prevent self-updates.

If the values of runtime data objects change often the feature can get performance intensive. As a result, the feature is disabled by default in simulator's settings. The settings panel is shown in figure 30. The settings panel and settings system were introduced for this purpose. Simulator stores settings in the web browser's local storage.

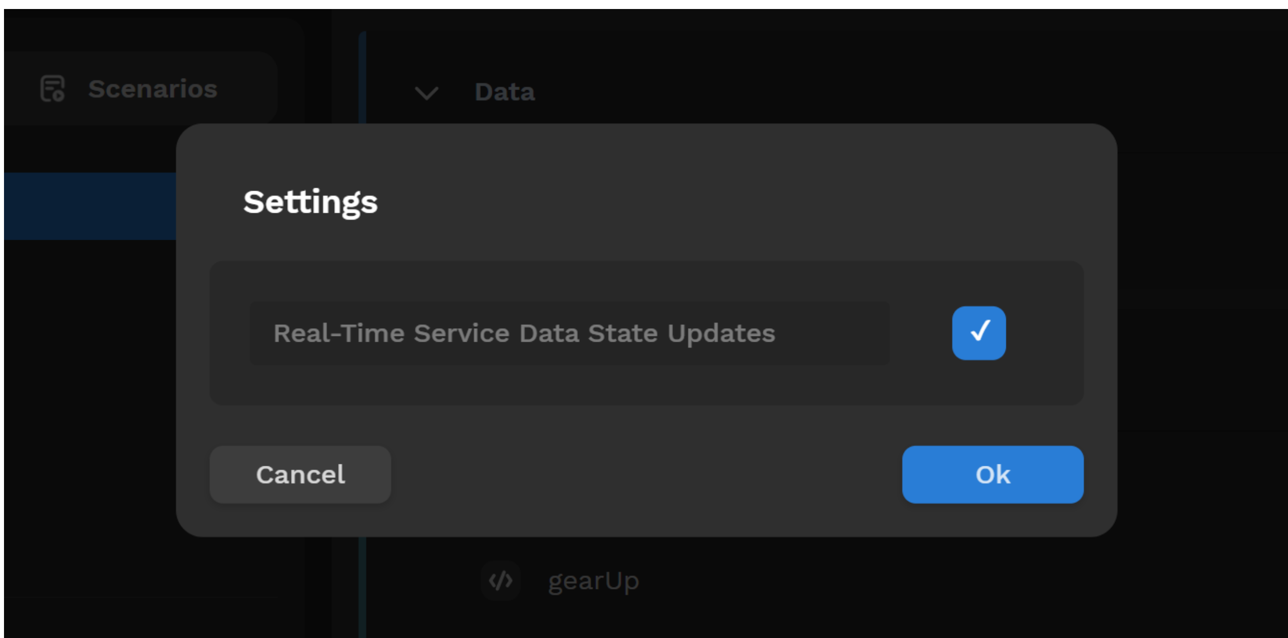

Figure 30. Settings panel in Kanzi Connect Simulator

Simulator uses the "/events" endpoint to listen to service events from the server. This endpoint is used to implement a feature where simulator is able to automatically add and remove services from its state as services are added or removed from the server. Simulator listens for the service register and unregister events. If a service is registered and simulator is not aware of the service, the service is added to simulator's state. If a service is unregistered and simulator is aware of the service, the

service is removed from simulator's state. This is especially useful when working with remote services that are often dynamically registered to the server after the initial state synchronization in simulator. Figure 31 shows example notifications for service register and unregister events.

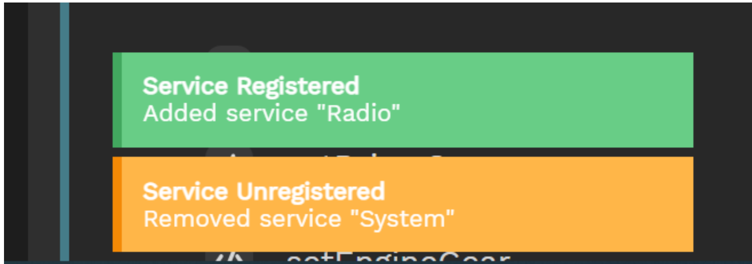

Figure 31. Service register and unregister notifications in Kanzi Connect Simulator

Code block 21 shows the handling of WebSocket events. The event handler has a similar problem to the runtime data handler where simulator gets updates from its own actions which it is unable to filter out. When an end user adds a service to the server using the simulator user interface, the server-side operation is attempted and verified first and only then simulator adds the service to its own state. The problem is that on success the WebSocket event notifying simulator of service registration comes back to simulator before simulator has had time to add the service to its state. Because simulator cannot identify who caused the service registration event, it assumes that the service it just itself created was added to the server by someone else. The result is that the service is added to simulator's state through the wrong code path and the user sees a notification that a service was registered. The same problem applies to service unregister events. The solution is that simulator waits for a short time period before processing these events. This way simulator has had time to add the service to its state and the self-update events are effectively ignored.

```
eventWebSocket.onmessage = async message => {
    message = JSON.parse(message.data)
    if (message.event == "serviceRegistered") {
        await wait(100)
        const serviceName = message.args[0]
        await dispatch("services/onServiceRegistered", { serviceName: serviceName }
, { root: true })
    }
    else if (message.event == "serviceUnregistered") {
        await wait(100)
        const serviceName = message.args[0]
        await dispatch("services/onServiceUnregistered", { serviceName: serviceName
 }, { root: true })
    }
}
```

Code Block 21. Handling WebSocket events

7.2 HTTP API

This chapter evaluates the Kanzi Connect HTTP API. The API is first evaluated from a high-level perspective to get an overview. The APIs of the ServiceManager service and the Scenario service are then evaluated. These are the two most important services from the perspective of Kanzi Connect Simulator. The API is not modified in practice due to time constraints and the effect this would have on existing systems that use the API.

7.2.1 Overview

The HTTP API is essentially an RPC API. The API provides a generic way of invoking arbitrary service methods over HTTP. The API uses only the GET and POST HTTP methods. These two HTTP methods can be used interchangeably because the server does not know which methods represent read-only operations. The endpoint URI contains the executed operation. Code block 22 shows examples of endpoints that invoke methods on the Persistence service.

```
// Read a setting value.
POST /serviceinvoke/persistence/method/readSettingValue

// Write a setting value.
POST /serviceinvoke/persistence/method/writeSettingValue

// Delete a setting.
POST /serviceinvoke/persistence/method/deleteSetting
```

Code Block 22. Persistence service HTTP API endpoints

There is also a technical reason for using RPC instead of REST. Service methods are defined in service definition files. A service definition file does not allow multiple method elements to have the same name and there is no mechanism for specifying HTTP methods. The HTTP API is also not the only or even the primary way of invoking service methods. Applications developed with Kanzi Studio are likely to use some other mechanism. It could be possible to extend the service definition and the server-side functionality to allow defining of REST-like endpoints. Code block 23 shows an example of a service definition extended in this way and the resulting endpoints.

```
// Extended service description.
<service name="example">
    <method name="createUser">
        <rest-endpoint name="/users" method="POST" />
    </method>

    <method name="updateUser">
        <rest-endpoint name="/users/:id" method="PUT" />
    </method>

    <method name="deleteUser">
        <rest-endpoint name="/users/:id" method="DELETE" />
    </method>

    <method name="someOperation">
        <rest-endpoint name="/someOperation" method="POST" />
    </method>
</service>


// Resulting endpoints.

// Create user.
POST /services/example/users

// Update user.
PUT /services/example/users/{id}

// Delete user.
DELETE /services/example/users/{id}

// Some (non-CRUD) operation.
POST /services/example/someOperation
```

Code Block 23. Extended service definition and the resulting endpoints

REST endpoints as defined in the previous example might add some value to services that have many CRUD methods. The existing RPC API however seems like a good fit for the use case where arbitrary methods are called. The ability to define parallel REST endpoints would add a lot of complexity to the system for questionable benefit. The API as defined in the previous example would also still not be a true REST API. The Richardson maturity model requires that level three compliant REST APIs use dynamic links. It is also unclear how methods would respond with different HTTP status codes.

7.2.2 Service API

Kanzi Connect Simulator uses the API of the (confusingly named) “Service” service to modify services. The service manages services running on a Kanzi Connect server. For clarity the service will be referred to as the ServiceManager service which is also the name of the concrete C++ class for the service. This chapter evaluates the API of the ServiceManager service for a subset of methods that are used by Kanzi Connect Simulator.

The first set of methods is used to create services, delete services, and set service attributes. Code block 24 shows these endpoints. The endpoints to create and delete a service invoke a multi-purpose method called “controlService”.

```
// Create a service.
POST
/serviceinvoke/service/method/controlService?id=&action=create&targetname={serviceN
ame}&argument={serviceDefinition}}

// Delete a service.
POST /serviceinvoke/service/method/controlService?id={serviceName}&action=destroy&t
argetname={serviceName}&argument=none

// Set a service attribute.
POST /serviceinvoke/service/method/setServiceAttribute?serviceIdentifier={serviceNa
me}&attributeName={name}&attributeValue={value}
```

Code Block 24. ServiceManager service API for working with services

The endpoints in the previous code block could be simplified by having a separate method for each operation. The endpoint to create a service could be split into two methods: one for creating a new service and one for importing an existing service. The method to set a service attribute could be a generic method that all services have. Code block 25 shows the modified endpoints.

```
// Create a service.
POST /serviceinvoke/service/method/createService?name={serviceName}

// Delete a service.
POST /serviceinvoke/service/method/deleteService?name={serviceName}

// Set a service attribute.
POST /serviceinvoke/{service}/method/setAttribute?name={name}&value={value}

// Import a service.
POST /serviceinvoke/service/method/importService
{serviceDefinition}
```

Code Block 25. Modified ServiceManager service API for working with services

The ServiceManager service has methods for working with every service element type: data, methods, events, properties, enumerations, types, and routes. Code block 26 shows an example of the endpoints for working with service event elements. The endpoints for other service elements follow a similar style.

```
// Create an event element.
POST
/serviceinvoke/service/method/addEvent?serviceIdentifier={serviceName}&eventIdentif
ier={eventName}&xml={xml}

// Delete an event element.
POST /serviceinvoke/service/method/removeEvent?serviceIdentifier={serviceName}&even
tIdentifier={eventName}
```

Code Block 26. ServiceManager service API for service event elements

The first observation is that there is no method to update an event element. If a consumer of the API wants to update an event element it has to first delete the existing event element and then create the updated event element on the server. This applies to all service element types. The problem with this is that the API moves responsibility of data integrity to consumers of the API. If an element that is updated in this way is referenced by other elements in the service definition, the consumer of the API is responsible for updating the related elements. For example, a type element defines a custom data type in a service definition. Other elements in a service definition such as data and method elements can use the data type in their definitions. If the type element is renamed

by deleting it and recreating it, the server does not update the elements that refer to that type to use the new type name. If a type element is referred to by 10 other elements in a service definition, the consumer has to send 22 API requests to keep the service definition valid. That is two API requests to update the type and 20 API requests to update the 10 elements that use the type. This operation should only require one API request to a method that updates the type element. The lack of methods to update elements simplifies the API implementation but moves the complexity of updating elements to the consumer of the API. This makes it difficult to have different clients that use the API because every client would have to implement the required functionality to keep the service definition valid. The solution would be to move responsibility of data integrity to the API implementation and extend the API with update methods.

To create an element the consumer of the API has to serialize the element in the XML format used in a service definition. The XML must also be encoded in Base64. The encoded XML is not human-readable which makes it inconvenient to use or test the API using a desktop application such as Postman or a command-line tool such as curl. Postman and curl are both tools that can be used to send HTTP requests (Postman 2021; Curl 2021). It could be argued that the API exposes implementation details by using the service definition XML in the API endpoints. Consumers of the API have to be able to serialize and deserialize the XML which makes it difficult to modify the API implementation without affecting clients. If the data format used in the API differed from the format used to represent service definitions internally, the API implementation could evolve independently from its consumers.

The findings indicate that the ServiceManager service API has been designed from the perspective of the API provider. As described in chapter six, an API should be designed from the perspective of the API consumer. This would result in an API that is easier to use and help in identifying the goals that consumers of the API want to achieve.

7.2.3 Scenario API

Kanzi Connect Simulator uses the API of the Scenario service to work with scenarios. This chapter evaluates the API of the Scenario service. As described in chapter three, scenarios allow services to be controlled with JavaScript scripts that are executed on the server. A scenario consists of one or more scripts, which all have one or more triggers that determine when the script is executed.

The first set of methods is used to create and delete scenarios, and to start and stop scenarios. Code block 27 shows these endpoints. The endpoints are simple from the perspective of the API consumer. The only complexity is in the method to add a scenario which requires the consumer to serialize the scenario schema in XML format and encode it in Base64 similar to the ServiceManager service API.

```
// Create a scenario.
POST /serviceinvoke/scenario/method/addScenario?name={name}&schema={schema}

// Delete a scenario.
POST /serviceinvoke/scenario/method/removeScenario?name={name}

// Start a scenario.
POST /serviceinvoke/scenario/method/startScenario?name={name}

// Stop a scenario.
POST /serviceinvoke/scenario/method/stopScenario?name={name}
```

Code Block 27. Scenario service API for working with scenarios

The second set of methods is used to add individual scripts and triggers to scenarios. These endpoints are not currently used by Kanzi Connect Simulator. The workflow in Kanzi Connect Simulator is to make changes to a scenario locally and then deploy the full scenario using the method to add a scenario. This allows a previous functional version of the scenario to be executed on the server while the scenario is modified locally. Code block 28 shows these endpoints.

```
// Add a script.
POST
/serviceinvoke/scenario/method/addLambda?name={name}&definition={definition}&servic
es={services}

// Remove a script.
POST /serviceinvoke/scenario/method/removeLambda?name={name}

// Add a data trigger.
POST /serviceinvoke/scenario/method/addDataBinding?name={name}&lambda={lambda}&sour
ce={source}&initialDelay={initialDelay}&interval={interval}&repeats={repeats}

// Add an event trigger.
POST /serviceinvoke/scenario/method/addEventBinding?name={name}&lambda={lambda}&sou
rce={source}&initialDelay={initialDelay}&interval={interval}&repeats={repeats}

// Add a timer trigger.
POST /serviceinvoke/scenario/method/addTimerBinding?name={name}&lambda={lambda}&ini
tialDelay={initialDelay}&interval={interval}&repeats={repeats}

// Remove a trigger.
POST /serviceinvoke/scenario/method/removeBinding?name={name}
```

Code Block 28. Scenario service API for working with scenario components

The observation here is that the API endpoints and Kanzi Connect Simulator use different terminology. The API endpoints refer to scripts and triggers as lambdas and bindings. The same inconsistency can be observed in the scenarios end user documentation where both versions are used interchangeably. There is a possibility that the API endpoints have been designed from the perspective of the API provider and that they are exposing API implementation details. The API likely implements triggers as bindings to service elements, but on the consumer's side trigger seems more self-explanatory as they are used to define when a script is executed. In programming lambda usually refers to an anonymous function whereas scenario scripts have names. Regardless of which is correct, the fact that the API, end user documentation, and Kanzi Connect Simulator all use different terminology is inconsistent and confusing to the API consumer.

8 Output

The output of the development is a WebSocket API and its related client-side functionality. The WebSocket API has been integrated to the Kanzi Connect master branch. The client-side functionality that uses the WebSocket API endpoints has been integrated to the Kanzi Connect Simulator master branch. Kanzi Connect and Kanzi Connect Simulator have their own Git repositories, version numbers, and releases, but in practice Kanzi Connect Simulator is publicly released as part of a Kanzi Connect release. The client-side functionality was released internally as part of Kanzi Connect Simulator version 2.1, which will be publicly released as part of Kanzi Connect version 2.0 later this year.

As stated in the previous chapter the HTTP API was not modified in practice. The results of the evaluation may however be used to guide future development of the API. In addition to the planned functionality the development produced features that may be used for other purposes. These include for example the application settings system in Kanzi Connect Simulator, and the ability to subscribe to messages using regular expressions in Kanzi Connect server-side code. All of the developed software is proprietary, and as such the development did not result in contributions to the open-source community in the form of software libraries or components.

The developed functionality has been used and tested internally, and it has proved effective at solving the identified problems. Kanzi Connect Simulator is now able to reliably determine the state of the Kanzi Connect server and react accordingly without false negatives. Use of WebSocket has also reduced flakiness in simulator's end-to-end tests and improved use of simulator with a slow network connection. The WebSocket events endpoint allows simulator to automatically add and remove services from its state as services are registered and unregistered from the server. The WebSocket data endpoint allows simulator to display the state of service runtime data in real-time. An edge case that was left unresolved is that updates to the values of remote service runtime data values do not go through the WebSocket API. Remote services run in a separate process outside the server.

Both Kanzi Connect and Kanzi Connect Simulator have automated test suites. The end-to-end tests of Kanzi Connect Simulator partially cover testing of the WebSocket functionality, but overall, automated testing of the WebSocket API is still lacking. Any problems in the WebSocket functionality would however quickly become apparent due to the integral role it now has in the operation of Kanzi Connect Simulator. Nevertheless, this is an area that should be improved in the future.

9 Ethics

Specific ethical codes of conduct can be found in many professions including medicine, law, and engineering. The purpose of these ethical codes is to protect the clients, safeguard the practitioners, and ensure the good name of the profession. Software engineering does not have such universal rules. Few industry standards exist that software engineers can be usefully accredited against. Various organizations such as ACM have published their own codes of ethics, but these have little legal standing and are not universally recognized. (Goodliffe 2014, 243-244.)

The IEEE-CS/ACM joint task force on Software Engineering Ethics and Professional Practices (SEEPP) has defined a code of ethics for software engineers. The code defines eight principles that software engineers should adhere to in accordance with their commitment to the health, safety, and welfare of the public. The code states that software engineers should commit themselves to making the analysis, specification, design, development, testing, and maintenance of software a beneficial and respected profession. (IEEE-CS/ACM 1999.)

The full version of the IEEE-CS/ACM code of ethics gives examples and details on the aspirations. The short version of the code defines the eight principles as follows:

1. *PUBLIC – Software engineers shall act consistently with the public interest.*

2. *CLIENT AND EMPLOYER – Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.*

3. *PRODUCT – Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.*

4. *JUDGMENT – Software engineers shall maintain integrity and independence in their professional judgment.*

5. *MANAGEMENT – Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.*

6. *PROFESSION – Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.*

7. *COLLEAGUES – Software engineers shall be fair to and supportive of their colleagues.*

8. *SELF – Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession. (IEEE-CS/ACM 1999)*

The goal with the thesis was to maintain a neutral viewpoint in the research and the development and evaluation of the software. The author is an employee of Rightware responsible for the development of Kanzi Connect Simulator, which has the potential to introduce a certain bias. The HTTP API in particular was evaluated from the perspective of a consumer of the API. The evaluation likely does not consider all the technical details and limitations that may have influenced the API implementation. An outside entity or someone primarily developing the Kanzi Connect server-side code may have a different perspective on the topic. However as described in chapter six, an API should be designed from the perspective of the API consumer. As such it is unlikely that there are significant problems with the evaluation, but it is good to keep in mind that what was written is not the full story.

The software was developed to the author’s best ability following the team’s processes, practices, and coding guidelines. The developed functionality was peer code reviewed and run through a continuous integration pipeline before getting merged into the main codebase. The peer code review for the client-side code was not as comprehensive as for the server-side because the author is the sole software engineer responsible for the development of Kanzi Connect Simulator. The software uses external libraries and frameworks according to their licensing terms and gives appropriate attribution. The software was developed using the most appropriate tools and techniques with the aim of producing the highest quality product possible. The software was written with the intention of providing value and improving the lives of the end users and other stakeholders.

10 Discussion

The API evaluation focused on the overall design and areas most relevant to Kanzi Connect Simulator. The evaluation could have gone into more detail and also covered areas such as documentation, testing, and security. The original assumption was that the API could be evaluated using REST constraints. This was due to unfamiliarity with REST and the seemingly common misconception that REST is the superior API style on the Web. It often seems as though every Web API should strive to be a REST API if only to be called "REST-like". RPC, REST, and other API styles are all valuable depending on the context (Lauret 2015; Doerrfield 2018). The Kanzi Connect HTTP API is an example of an API where RPC is a natural fit. The evaluation of the ServiceManager and Scenario service APIs demonstrated the importance of the API design guidelines described in chapter six. The thesis did not cover the design of event-driven APIs in particular. The exchanged data and interactions with the WebSocket API are fairly simple for now, but this is an area that should be explored more if the WebSocket API is extended in the future. The API landscape is vast and ever-changing, and this thesis only scratched the surface. It is difficult to give conclusive statements regarding technology choice, but the old adage of trying to use the right tool for the right job should apply here as well.

An alternative perspective for web-based communication in Kanzi Connect would be the link between the Kanzi Connect server and the Kanzi Connect tools for Kanzi Studio. Web-based communication is used there for example to import services from the server to Kanzi Studio. Kanzi Studio presumably polls the server to determine whether the interfaces of imported services have changed, and then fetches the updated service definitions from the server. An alternative approach would be to listen for WebSocket events that indicate changes to service definitions, and then fetch the updated services. This would eliminate the need for polling the HTTP API. This same idea could be used in Kanzi Connect Simulator to enable multi-user support. Currently only one user or simulator instance can reliably edit services on the same server. Simulator does not monitor the server for changes to service definitions, and changes from one simulator instance do not automatically propagate to other simulator instances. Simulator could similarly listen for WebSocket events that indicate changes to service definitions, and then fetch the updated services. There may be some details that are not considered, but this would go a long way towards allowing multiple users to edit services on the same server.

An objective that was left out from the thesis is the ability to write service data values to the server using WebSocket. Currently simulator sets the values of service data elements using a service method in the HTTP API. The primary use case for this would have been to fix an issue with the slider controller. Changing the value of a service data element using a range input produces a large amount of HTTP requests to the server. The issue with this is two-fold. First, the values do not update evenly to the Kanzi Connect server and consequently to Kanzi Studio. Strangely enough this has only been observed with Chromium-based web browsers such as Google Chrome and Microsoft Edge. The second problem is that as described in chapter four, the flood of HTTP requests from the range input block the polling requests that are supposed to determine the state of the server. This resulted in false negatives which then caused problems with the state synchronization. The WebSocket API fixes the latter issue, but the problem of uneven updates with a range input remains. This is not a high priority issue, but still something that could be looked into in the future.

A feature related to web application communication that could be implemented in the future is the ability for Kanzi Connect Simulator to connect to arbitrary or multiple Kanzi Connect servers. Currently the Kanzi Connect server hosts the simulator web application, and simulator can only communicate with the server that is hosting it. This has sometimes caused problems when the server has been running on a mobile or embedded device. To have access to simulator the Kanzi Connect server must be deployed together with the embedded CivetWeb web server's root directory. The root directory is referred to as the content pack, which among other things includes the simulator web application. Deploying the content pack can sometimes be problematic, or it may not be desirable to deploy it on a device with low technical specifications. The solution would be to allow the end user of simulator to specify the Kanzi Connect server that simulator is communicating with. This would allow simulator to communicate with a server that is running on a separate computer without deploying the content pack. The feature could also be used to make it easier to work with multiple Kanzi Connect servers by allowing simulator to store the connection information of multiple servers. This may however require an alternative local or cloud-based web server for hosting the simulator web application. Hosting Kanzi Connect Simulator in the cloud is another interesting network related topic to consider. Cloud hosting would allow Kanzi Connect Simulator to be released independently of Kanzi Connect and enable continuous deployment where updates to simulator would be immediately available to users.

The Kanzi Connect Simulator user interface is fully responsive which means that it scales and adjusts its layout to fit the screen or window size that it is viewed on. Figure 32 shows simulator running on a Google Pixel XL phone. A constraint that sometimes comes with mobile devices is a slow network connection. This has been an uncommon use case so far, but simulator's performance in such an environment could be investigated more. Operations that are more data-intensive such as synchronizing state and fetching service definitions may take a long time to complete. This is another scenario where the WebSocket API could prove useful in improving performance.

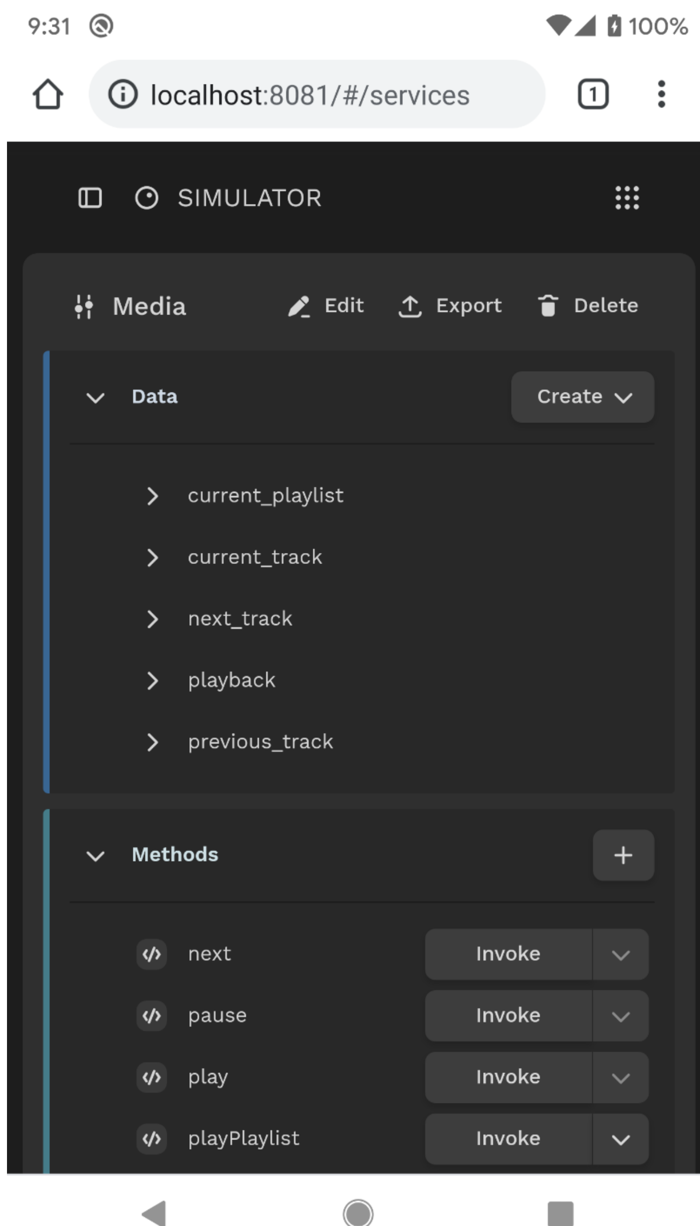

Figure 32. User interface responsiveness in Kanzi Connect Simulator

Kanzi Connect Simulator was extended with a diagnostics view during the writing of the thesis. The diagnostics view is shown in figure 33. It was released as part of Kanzi Connect Simulator version 2.1. The diagnostics view currently polls the server HTTP API to get the updated data. This is very inefficient especially in the case of the server log as the server may go long periods of time without writing anything to its log. A better approach would be to extend the WebSocket API with endpoints that provide the diagnostics data. This way the data would be immediately available and polling the HTTP API would not be needed.

DIAGNOSTICS KANZI CONNECT

Server Log All

```
2021-08-25T02:40:06  INF  virtualfile_service    Locally registered path: http://<server>:8080/contentproviderdata/
2021-08-25T02:40:06  INF  virtualfilerequesth…   Created Local handler for path '/contentproviderschema/'
2021-08-25T02:40:06  INF  virtualfile_service    Locally registered path: http://<server>:8080/contentproviderschema/
2021-08-25T02:40:06  INF  virtualfilerequesth…   Created Local handler for path 'data://contacts/'
2021-08-25T02:40:06  INF  virtualfile_service    Locally registered path: data://contacts/
2021-08-25T02:40:06  INF  virtualfilerequesth…   Created Local handler for path 'data://albumart/'
2021-08-25T02:40:06  INF  virtualfile_service    Locally registered path: data://albumart/
2021-08-25T02:40:07  INF  content_manager        Creating content provider to handle URI com.rightware.content
2021-08-25T02:40:07  INF  content_manager        Creating content provider to handle URI com.rightware.diagnostics
2021-08-25T02:40:10  INF  service_manager        First session for 'Media' -> About to start service.
2021-08-25T02:40:10  INF  media_service          onStartServiceRequest called.
```

Server Monitor Services

Name	Instances	Location	RX/TX Speed	RX/TX Bytes	RX/TX Messages
Virtualfile	1	local	0/0 kbps	0/0	0/0
System	1	local	0/0 kbps	0/0	0/0
ServiceInvoke	1	local	0/0 kbps	0/0	0/0
Service	1	local	0/0 kbps	0/0	0/0
Sensor	1	local	0/0 kbps	0/0	0/0
Scenario	1	local	0/0 kbps	0/0	0/0
Persistence	1	local	0/0 kbps	0/0	0/0
Obd2	1	local	0/0 kbps	0/0	0/0
Media	1	local	0/0 kbps	0/0	0/0
Input	1	local	0/0 kbps	0/0	0/0
Diagnostics	1	local	0/0 kbps	0/0	0/0

Figure 33. Diagnostics view in Kanzi Connect Simulator

The introduction of the diagnostics view in version 2.1 of Kanzi Connect Simulator also turned simulator into more of a collection of applications. This is illustrated in figure 34. The change resulted in another set of challenges in optimizing web application communication. For example, the server monitor component should only poll the server for new data when the diagnostics view is visible. This is because it shows the current state of the server and not historical data. The server log component however has to poll the server at all times because the HTTP API endpoint only returns the last N log messages. By default, the server stores the last 75 log messages internally. This means that the server log component must poll the HTTP API to keep track of the full server log over time. The server log component should also be cleared only when the server changes state from offline to online. This way the log messages can still be viewed and analyzed even if the server shuts down or crashes.

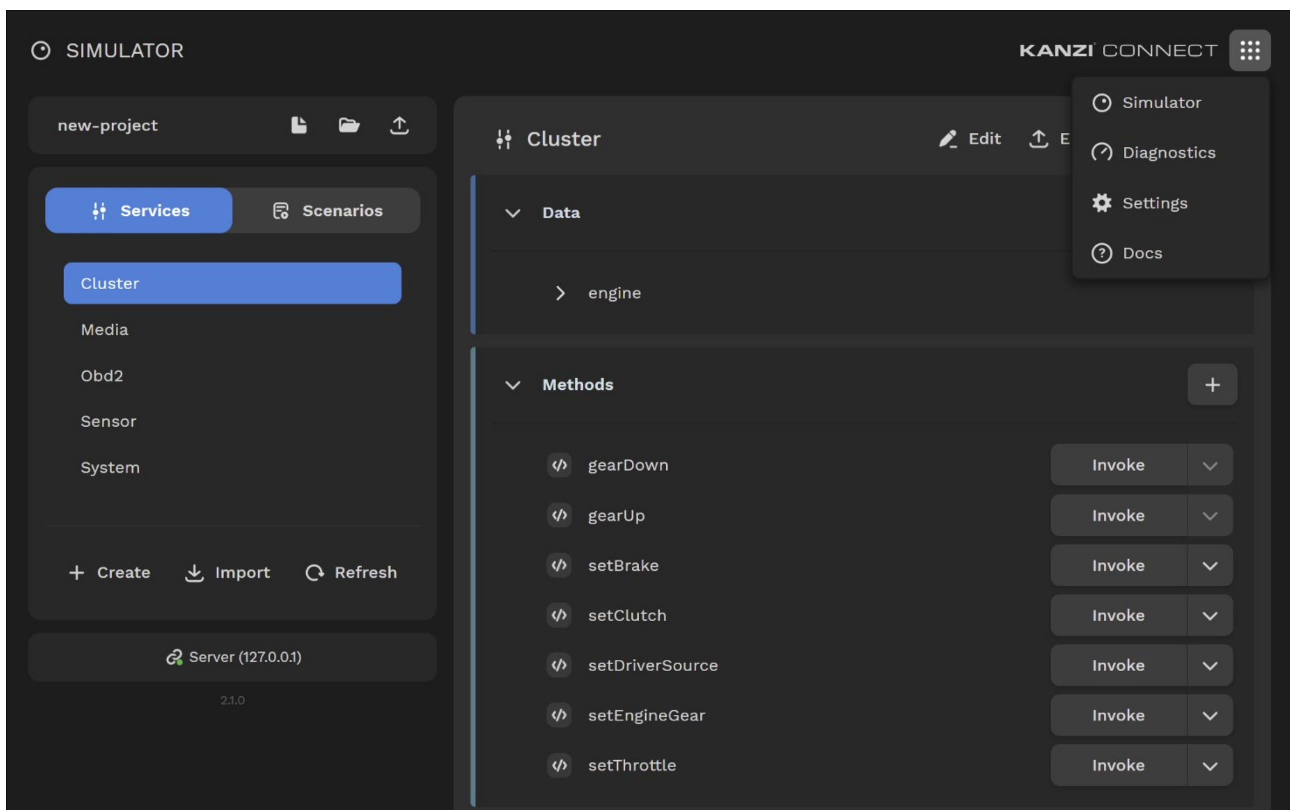

Figure 34. Version 2.1 of Kanzi Connect Simulator

All in all, Kanzi Connect and Kanzi Connect Simulator have taken a long stride forward in web application communication with the introduction of the WebSocket API. But as the discussion shows, there is still a lot that can be done.

11 Conclusion

The thesis improved web-based communication with a Kanzi Connect server by implementing a WebSocket API. The thesis improved Kanzi Connect Simulator in terms of web application communication by implementing new features and resolving existing problems using the WebSocket API. The thesis also evaluated the Kanzi Connect HTTP API and proposed improvements based on the theoretical framework.

The thesis developed a new construct in the form of the WebSocket API and its related client-side functionality. The construct is functional and solves the practical research objectives, but its theoretical contribution is mostly as a practical demonstration of existing theoretical knowledge. The implemented construct itself does not have much use outside of Kanzi Connect, but the more general concept of sending system events or updates through an event-driven API may well be applicable elsewhere.

The existing HTTP API was not modified in practice, but its evaluation can be used in the future development of the API. The research findings recognized the value that different API styles have depending on the context. The evaluation also demonstrated the importance of following API design guidelines, and in particular the importance of designing an API from the perspective of the API consumer.

Web application communication in Kanzi Connect will continue to be improved as new requirements emerge. As the previous chapter demonstrated, web application communication in Kanzi Connect and Kanzi Connect Simulator can still be improved in a number of ways. The role of web application communication in production use of Kanzi Connect is unclear, but it is an area that could substantially increase the requirements in the future.

References

CivetWeb. (2021). Embedding CivetWeb. Retrieved 2021-07-21 from https://github.com/civetweb/civetweb/blob/master/docs/Embedding.md

Curl. (2021). Index. Retrieved 2021-08-06 from https://curl.se/

Doerrfield, B. (2018). Is REST Still a Relevant API Style? Retrieved 2021-08-15 from https://nordicapis.com/is-rest-still-a-relevant-api-style

Fielding, R. (2000). Architectural Styles and the Design of Network-based Software Architectures. Retrieved 2021-07-23 from https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

Fielding, R. (2008). REST APIs must be hypertext-driven. Retrieved 2021-08-07 from https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

Fielding, R., & Reschke, J. (2014). Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. Retrieved 2021-07-19 from https://datatracker.ietf.org/doc/html/rfc7231

Fowler, M. (2010). Richardson Maturity Model. Retrieved 2021-07-25 from https://martinfowler.com/articles/richardsonMaturityModel.html

GitHub Docs. (2021a). GitHub REST API. Retrieved 2021-08-02 from https://docs.github.com/en/rest

GitHub Docs. (2021b). GitHub GraphQL API. Retrieved 2021-08-02 from https://docs.github.com/en/graphql

GitHub Docs. (2021c). Forming calls with GraphQL. Retrieved 2021-08-02 from https://docs.github.com/en/graphql/guides/forming-calls-with-graphql

Goodliffe, P. (2014). Becoming a Better Programmer. Sebastopol, CA: O'Reilly Media.

Grigorik, I. (2013). High-Performance Browser Networking. Sebastopol, CA: O'Reilly Media.

IBM Cloud Education. (2021). REST APIs. Retrieved 2021-07-23 from https://www.ibm.com/cloud/learn/rest-apis

IEEE-CS/ACM. (1999). Code of Ethics. Retrieved 2021-08-25 from https://www.computer.org/education/code-of-ethics

Jin, B., Sahni, S., & Shevat, A. (2018). Designing Web APIs. Sebastopol, CA: O'Reilly Media.

Kanzi Documentation. (2021a). Kanzi Connect Overview. Retrieved 2021-02-13 from https://docs.kanzi.com/connect/1.1.2/overview.html

Kanzi Documentation. (2021b). Kanzi Connect fundamentals. Retrieved 2021-02-13 from https://docs.kanzi.com/connect/1.1.2/architecture/kanzi-connect-architecture.html

Kanzi Documentation. (2021c). Using the HTTP API. Retrieved 2021-02-23 from https://docs.kanzi.com/connect/1.1.2/working/httpapi/httpapi.html

Kanzi Documentation. (2021d). Kanzi Connect Simulator. Retrieved 2021-02-23 from https://docs.kanzi.com/connect/1.1.2/working/simulator/simulator.html

Kanzi Documentation. (2021e). Kanzi Connect services. Retrieved 2021-02-24 from https://docs.kanzi.com/connect/1.1.2/architecture/kanzi-connect-services.html

Kanzi Documentation. (2021f). Core services. Retrieved 2021-02-24 from https://docs.kanzi.com/connect/1.1.2/architecture/core-services.html

Kanzi Documentation. (2021g). Scenarios. Retrieved 2021-04-20 from https://docs.kanzi.com/con-nect/1.1.2/working/simulator/scenarios.html

Lauret, A. (2015). Do you really know why you prefer REST over RPC? Retrieved 2021-08-15 from https://apihandyman.io/do-you-really-know-why-you-prefer-rest-over-rpc/

Lauret, A. (2019). The Design of Web APIs. Shelter Island, NY: Manning Publications Co.

Lukka, K. (2014). Konstruktiivinen tutkimusote. Retrieved 2021-03-08 from https://metodix.fi/2014/05/19/lukka-konstruktiivinen-tutkimusote/

MDN Web Docs. (2021a). Fetch API. Retrieved 2021-05-31 from https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

MDN Web Docs. (2021b). HTTP. Retrieved 2021-07-18 from https://developer.mozilla.org/en-US/docs/Web/HTTP

MDN Web Docs. (2021c). An overview of HTTP. Retrieved 2021-07-18 from https://devel-oper.mozilla.org/en-US/docs/Web/HTTP/Overview

MDN Web Docs. (2021d). HTTP request methods. Retrieved 2021-07-19 from https://devel-oper.mozilla.org/en-US/docs/Web/HTTP/Methods

MDN Web Docs. (2021e). HTTP response status codes. Retrieved 2021-07-19 from https://devel-oper.mozilla.org/en-US/docs/Web/HTTP/Status

MDN Web Docs. (2021f). Identifying resources on the Web. Retrieved 2021-07-19 from https://de-veloper.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Identifying_resources_on_the_Web

MDN Web Docs. (2021g). XMLHttpRequest. Retrieved 2021-07-20 from https://devel-oper.mozilla.org/en-US/docs/Web/API/XMLHttpRequest

MDN Web Docs. (2021h). Using XMLHttpRequest. Retrieved 2021-07-20 from https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/Using_XMLHttpRequest

MDN Web Docs. (2021i). Fetching data from the server. Retrieved 2021-07-20 from https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Fetching_data#fetch

MDN Web Docs. (2021j). Using server-sent events. Retrieved 2021-07-20 from https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events/Using_server-sent_events

MDN Web Docs. (2021k). Writing WebSocket client applications. Retrieved 2021-07-21 from https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API/Writing_WebSocket_client_applications

MDN Web Docs. (2021l). Writing WebSocket servers. Retrieved 2021-07-21 from https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API/Writing_WebSocket_servers

Microsoft Docs. (2018). RESTful web API design. Retrieved 2021-08-07 from https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design

Nally, M. (2018). REST vs RPC: What problems are you trying to solve with your APIs? Retrieved 2021-08-01 from https://cloud.google.com/blog/products/application-development/rest-vs-rpc-what-problems-are-you-trying-to-solve-with-your-apis

Postman. (2021). Postman API Client. Retrieved 2021-08-06 from https://www.postman.com/product/api-client/

Red Hat. (2020). What is a REST API? Retrieved 2021-07-23 from https://www.redhat.com/en/topics/api/what-is-a-rest-api

Rightware. (2016). Thundersoft's acquisition of Rightware strengthens their leading positions in the fast-growing market for connected car software. Retrieved 2021-02-11 from https://www.rightware.com/blog/article/thundersofts-acquisition-of-rightware

Rightware. (2021a). About Rightware. Retrieved 2021-02-11 from https://www.rightware.com/company

Rightware. (2021b). Index. Retrieved 2021-02-13 from https://www.rightware.com/

Rightware. (2021c). Kanzi. Retrieved 2021-02-11 from https://www.rightware.com/kanzi

Rightware. (2021d). Kanzi Connect. Retrieved 2021-02-13 from https://www.rightware.com/kanzi-connect

Rightware. (2021e). Kanzi Reference HMI. Retrieved 2021-02-13 from https://www.rightware.com/kanzi-reference-hmi

Rightware. (2021f). References. Retrieved 2021-02-13 from https://www.rightware.com/kanzi/references

Rightware. (2021g). Solutions. Retrieved 2021-02-13 from https://www.rightware.com/solutions

Rightware. (2021h). Kanzi Workflow. Retrieved 2021-02-13 from https://www.rightware.com/kanzi/workflow

Rightware. (2021i). Rightware presents Innovation Award at Finland's Auto and Transportation Gala. Retrieved 2021-07-16 from https://www.rightware.com/blog/rightware-presents-innovation-award-at-finlands-auto-and-transportation-gala

Rightware. (2021j). Kanzi Reference HMI - accelerating digital cockpit development. Retrieved 2021-08-19 from https://www.rightware.com/blog/kanzi-reference-hmi-accelerating-digital-cockpit-development

Santoro, M., Vaccari, L., Mavridis, D., Smith, R. S., Posada, M., & Gattwinkel, D. (2019). Web Application Programming Interfaces (APIs): general-purpose standards, terms and European Commission initiatives. Retrieved 2021-07-25 from https://publications.jrc.ec.europa.eu/repository/handle/JRC118082

Slack API. (2021). Using the Slack Web API. Retrieved 2021-08-01 from https://api.slack.com/web

Stack Overflow. (2017). What is the difference between HTTP streaming and server sent events? Retrieved 2021-08-07 from https://stackoverflow.com/questions/42559928/what-is-the-difference-between-http-streaming-and-server-sent-events/42560354#42560354

Sturgeon, P. (2016). Understanding RPC vs REST for HTTP APIs. Retrieved 2021-08-01 from https://www.smashingmagazine.com/2016/09/understanding-rest-and-rpc-for-http-apis/

Sturgeon, P. (2017). A Response to REST is the new SOAP. Retrieved 2021-08-07 from https://phil.tech/2017/rest-confusion-explained/

Swagger. (2021a). OpenAPI Specification. Retrieved 2021-07-25 from https://swagger.io/specification/

Swagger. (2021b). Best Practices in API Design. Retrieved 2021-08-07 from https://swagger.io/resources/articles/best-practices-in-api-design/

Webber, J., Parastatidis, S., & Robinson, I. (2010). REST in Practice. Sebastopol, CA: O'Reilly Media.

WebRTC. (n.d.) Real-time communication for the web. Retrieved 2021-08-27 from https://webrtc.org/