



Thin proxies: REST APIs

Contents

Current status	2
System APIs	2
Application development	3
Proposal	4
REST APIs	4
REST APIs design	4
REST APIs security	5
REST APIs security for local services on embedded devices	7
Consuming REST APIs	7
Conclusion	8
Links	8

Apertis is a distribution that aims to provide solid bases to build products from IOT devices to complex HMI systems. The workflows to build such variety of products involves many different technologies, tools and developers'background.

A common issue during development is the need of high level APIs to interact with the system to allow application developers to focus on their use cases while hiding the complexity of low level system APIs.

Additionally, to hide complexity, a nice-to-have goal is to make these APIs technology agnostic, allowing developers to write their application in the language/technology they choose without any additional complexity.

An example of this situation is to let a developer with a basic network knowledge access a high level API to setup a simple network configuration in a HMI web application using Javascript. Unfortunately, currently, the only available solution to perform network configuration is to access the extremely complex **connman C D-BUS APIs**.

In order to provide a solution that satisfies these objectives, Apertis encourages the use of Thin Proxies.

Current status

System APIs

System APIs are used to allow application to interact with the system in order to:

- Retrieve information, such as retrieving network configuration
- Configure parameters, such as configuring network parameters
- Perform actions, such as system reboot

36 These kind of APIs are usually available in C language and bindings for other
37 languages are built on top them, such as Python, Go and Rust. In the case of
38 C++, usually the approach is to use the standard C API.

39 Since these kind of APIs are meant to cover many different use cases, they
40 usually provide low level functionality, making them extremely big and complex.
41 In addition they are very tied to specific technologies, requiring a deep knowledge
42 in order to properly use them.

43 Lastly, as it is clearly seen, the use of this kind of API imposes security risks
44 which should be minimized to provide a robust and reliable solution.

45 As a summary the challenges are:

- 46 • Usually built in the C language
- 47 • Provide low level functionality
- 48 • Very big and complex
- 49 • Tied to specific technologies
- 50 • Impose security risks

51 **Application development**

52 There is a wide range of applications which require access to system APIs to
53 fulfill their goals. However, it is very common to only need to use a small
54 number of high level operations. In such cases, accessing low level system APIs
55 as described previously represents a huge barrier for development due to:

- 56 • Big learning curve for system APIs
- 57 • Big learning curve for technologies involved
- 58 • No up to date binding for the language of preference
- 59 • Error prone due to limited experience

60 Additionally, it is common to build Flatpak applications, in order to provide an
61 easy way to distribute and upgrade confined applications, improving the security
62 and robustness of the solution. Under these premises, using system APIs directly
63 from Flatpak is not natural since it goes against the principle of application
64 confinement. To solve that Flatpak applications usually communicate with the
65 system services via D-Bus, but in some cases this is not ideal given it may
66 still require low-level knowledge of the components in question and is tied to a
67 specific IPC mechanism.

68 In these use cases a different approach should be to provide:

- 69 • Easy to use APIs
- 70 • Support for different languages/technologies
- 71 • Allow access from confined applications

72 **Proposal**

73 As a solution to overcome these difficulties Apertis encourages the use of Thin
74 Proxies, to provide easy to use high level APIs for system APIs. The idea behind
75 this concept consists in building a small service which provides an API targeted
76 to the specific use case to provide the required functionality. This service should
77 use REST APIs to provide a technology agnostic API.

78 There are other possible approaches, such as an IPC API which is used in other
79 projects. However, a REST API is much more technology agnostic, allowing
80 developers without experience in Linux systems to easily build applications.

81 **REST APIs**

82 A [REST API] uses HTTP to access resources using the standard methods
83 GET, PUT, POST and DELETE. This type of API is based in the concept
84 of representational state transfer (REST), with aims to allow scalability. The
85 goals behind using these kind of APIs are:

- 86 • Improve scalability of interactions between components
- 87 • Stateless operations
- 88 • Uniform interfaces
- 89 • Independent deployment of components
- 90 • Facilitate caching
- 91 • Enforce security
- 92 • Support layered system

93 For these reasons REST APIs are the most common technology to provide access
94 to remote resources on the Internet, allowing developers to build applications
95 in the language they prefer.

96 This same approach can be used to build applications that access local resources,
97 such as system APIs, using similar workflows than the ones used in other appli-
98 cations.

99 **REST APIs design**

100 Designing a REST API can be challenging since any change might impact in
101 the clients that make use of it. In this context the following recommendations
102 should help to reduce the possibility of having to deal with unexpected changes.

103 **Analyze use cases**

104 The first step in the process of designing an API is to understand what the
105 requirements of the use cases are. This step should also try to think about
106 possible new use cases with new requirements, to create an API that could be
107 extended naturally in new versions.

108 **Data format**

109 REST APIs can be used with different data such XML or JSON, however, nowa-
110 days is recommended to use JSON, since it is de-facto format for sending and
111 receiving data.

112 **URIs and endpoints**

113 Connected to the previous comment, designing URIs and endpoints, considering
114 current and possible future use cases will help developers using the REST API
115 when developing their applications and supporting it across new versions of the
116 API.

117 Additionally, when choosing names for endpoints it is recommended to use
118 *nouns*, since they represent objects, while the action on those objects is rep-
119 resented by the HTTP method used. In relation to this, make use of logical
120 nesting on endpoint to show relationships between them, making the API easier
121 to use.

122 **Error handling**

123 In order to make the API easier to use, *errors* should be handled gracefully and
124 standard HTTP codes, alongside additional text to describe the error, should
125 be returned when needed.

126 **Versioning**

127 Using versioning in the API ensures that the evolution of an API does not
128 affect old unsupported clients which are tied to an old version of the API while
129 allowing well supported clients to have access to the latest functionality.

130 **Documentation**

131 The process of documentation is important since this will allow to share the
132 design with the people involved, which will provide valuable feedback regarding
133 missing functionality or possible issues.

134 **REST APIs security**

135 As previously mentioned, one of the goals of this proposal is to provide a solution
136 that helps developers reduce the overhead without sacrificing security.

137 To do so, the same security principle that apply to any REST API on the
138 Internet also applies to Thin Proxies. These security principles can be relaxed
139 during development and testing but should be enforced in production.

140 **Block not allowed HTTP methods**

141 A common premise in security is to only allow the really needed functionality,
142 in order to reduce the possibility of exploits and attacks. With that in mind
143 only the HTTP methods that should be supported should be whitelisted, while
144 other methods should be blocked. As an example, the HTTP method DELETE
145 is usually not supposed to be used on common API calls and should be blocked.

146 **Use TLS and well supported security framework**

147 Nowadays, TLS is widely used on the Internet since it is the base for any se-
148 cure service. A very good practice is to make use of a well supported security
149 framework since this enables updates and security fixes to make the application,
150 in this case the Thin Proxy, more robust. By using TLS on REST APIs, both
151 confidentiality and server/service authentication are supported.

152 During development and testing the use of encrypted channels can be disabled
153 to help developers to debug their applications.

154 **Use solid authentication mechanism**

155 One of the most challenging aspects of security with REST APIs is to have a
156 solid authentication mechanism to prevent unauthorized access to the services
157 provided by the application.

158 The following are recommended mechanisms:

- 159 • [API keys](#)¹: solution based in the generation of a secret for the user, which
160 should be stored in the server/service.
- 161 • [OAuth](#)²: solution to pass authentication information from one service to
162 other, which offloads the authentication to a different service.
- 163 • [JSON Web Tokens \(JWT\)](#)³: solution based on token generation from
164 JSON data which are signed by the service.

165 **Data in URL**

166 It is discouraged to use sensitive data in the URL since this could lead to a
167 security leak. Data such as usernames, passwords and tokens should not be
168 included in the URLs since they could be easily captured by the server/service
169 logs or a network sniffer for example.

170 **Validate input parameters**

171 One way in which security can be compromised is by assuming that input data
172 is valid, since processing invalid data could lead to unexpected results. This
173 can happen either by an error in the application which is using the API or by
174 malicious users who are trying to get access to system resources.

175 By validating the input data the system will be able to reject API calls that use
176 invalid data minimizing the possibility of issues.

177 **Include timestamps**

178 Adding timestamps to HTTP request allows the server/service to check them
179 against the current time and in this way filter old requests preventing reply
180 attacks. This also adds value information to log records which could help during
181 debugging.

¹https://en.wikipedia.org/wiki/Application_programming_interface_key

²<https://en.wikipedia.org/wiki/OAuth>

³https://en.wikipedia.org/wiki/JSON_Web_Token

182 **Log failed requests**

183 Understanding what is happening is a key element to ensure security, and to
184 quickly respond to a security breach. In this context, having information related
185 to failed requests or abnormal situations, provides the data required to analyze
186 potential security issues.

187 **REST APIs security for local services on embedded devices**

188 The use of REST APIs is widely used to access services on the Internet but
189 as mentioned in this document the same principles can be used to access local
190 services in embedded devices. In this context, the same security principles apply
191 but some important differences need to be noted.

192 **Block requests from other hosts**

193 The fact that both client and server are in the same host allows to add addition-
194 ally restrictions, such as only allowing requests from local host. This restriction
195 could be relaxed on development environments to allow developers to easily test
196 their code.

197 **Solid authentication mechanism for embedded devices**

198 This document has covered some authentication mechanisms that are used on
199 the Internet, however not all of them are suitable for this use case. From the
200 previous list, JWT is the most suitable for this scenario, since the token is signed
201 with the service's private key and contains all the required information in the
202 token itself. In this way, the server/service does not need to know in advance
203 which clients will use it and does not need to store specific data for the client
204 application, which simplifies client deployments and avoid requiring a database.

205 **Consuming REST APIs**

206 The idea behind this document is to enable developers to consume the REST
207 APIs provided by Thin Proxies from the language/technology of their preference.
208 In this regard, only high level suggestions can be made that are tied to the
209 recommendations already suggested.

210 **Use TLS and well supported security framework**

211 As mentioned in the previous section, the use of TLS and well supported security
212 frameworks improves the security of the solution, since an attacker could make
213 use of a vulnerability in the consumer code to gain access to the system.

214 As commented before, during development and testing the use of encryption
215 can be disabled to help developers to debug their applications.

216 **Sensitive data**

217 Sensitive data such as username, JWT tokens or any other data that could give
218 useful information to attackers should be stored encrypted to minimize security

219 risks. Additionally, following the same principles, this type of data should not
220 be included in trace logs since it could be easily retrieved.

221 Conclusion

222 The use of Thin Proxies based on REST APIs provides a way for developers
223 to build their application using the language/technology they prefer, hiding the
224 complexity of low level APIs.

225 Links

- 226 • [REST API Best Practices⁴](#)
- 227 • [How to create a RESTful API⁵](#)
- 228 • [Securing REST APIs⁶](#)
- 229 • [How to secure REST API endpoints⁷](#)
- 230 • [REST API web service security⁸](#)

⁴<https://stackoverflow.blog/2020/03/02/best-practices-for-rest-api-design/>

⁵<https://yalantis.com/blog/how-to-create-a-restful-api/>

⁶<https://developer.okta.com/blog/2019/09/04/securing-rest-apis>

⁷<https://www.techtarget.com/searchcloudcomputing/tip/How-to-secure-REST-API-endpoints-for-cloud-applications>

⁸<https://www.netsparker.com/blog/web-security/rest-api-web-service-security/>