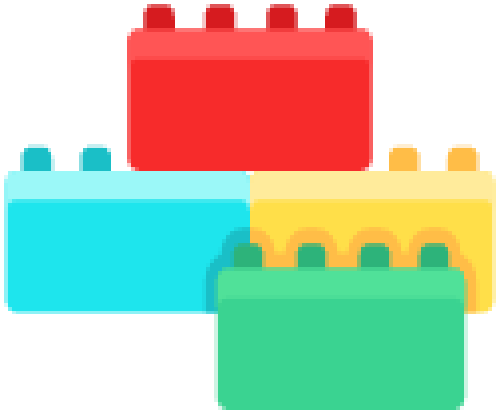


Bbriccs

gematik GmbH

Version 0.7.0, 2025-07-07

Contents

Scope

As a national competence center for digital health, gematik GmbH covers a broad spectrum of core areas of the German healthcare system. This includes the development of test suites, test tools and self-developed products for the digital health infrastructure.

Many products are built for TI, both inside and outside gematik GmbH. Redundant development is hard to avoid. However, the B²ric²s (`briks`) is trying to bundle many of the common solutions into a reusable SDK, which promises to significantly reduce development time and costs.

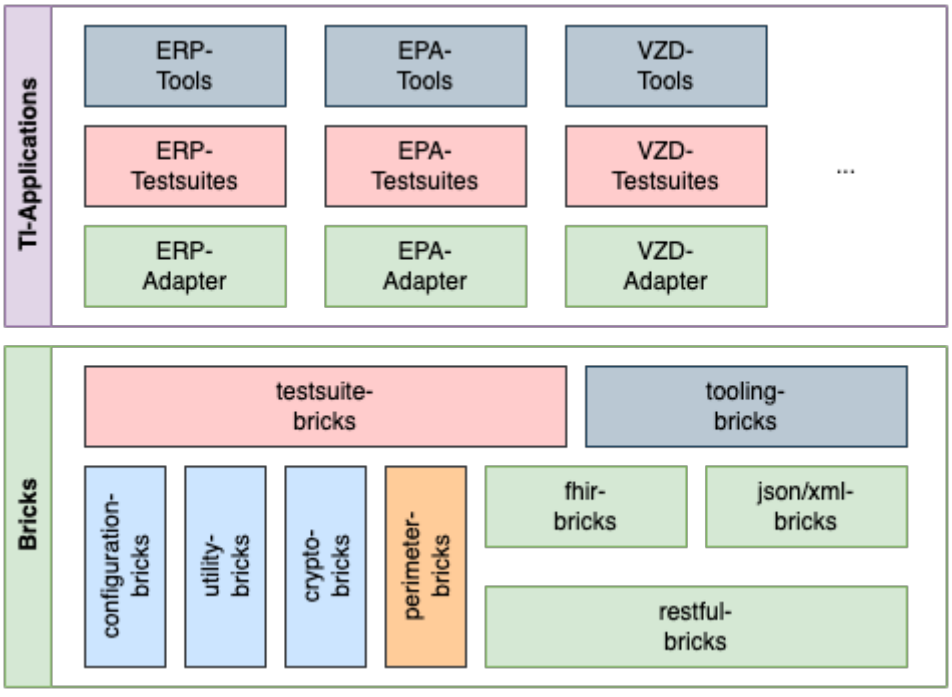
Acronym

While B²ric²s is just the word mark and is rather cumbersome in everyday use, the following uses are recommended depending on the situation:

- Pronunciation: simply `briks` based on the idea clamping blocks
- Spelling: in the simplest case just `bricks` or `bbriccs` as a mnemonic for the word mark
- Code: the spelling `bbriccs` is only to be used for package names and namespaces, while `brick` (or `bricks` for plural) shall be used for variable, class or module names

Bricks

The core objective of B²ric²s is to create a modular toolkit for more efficient development of applications for the German healthcare system.



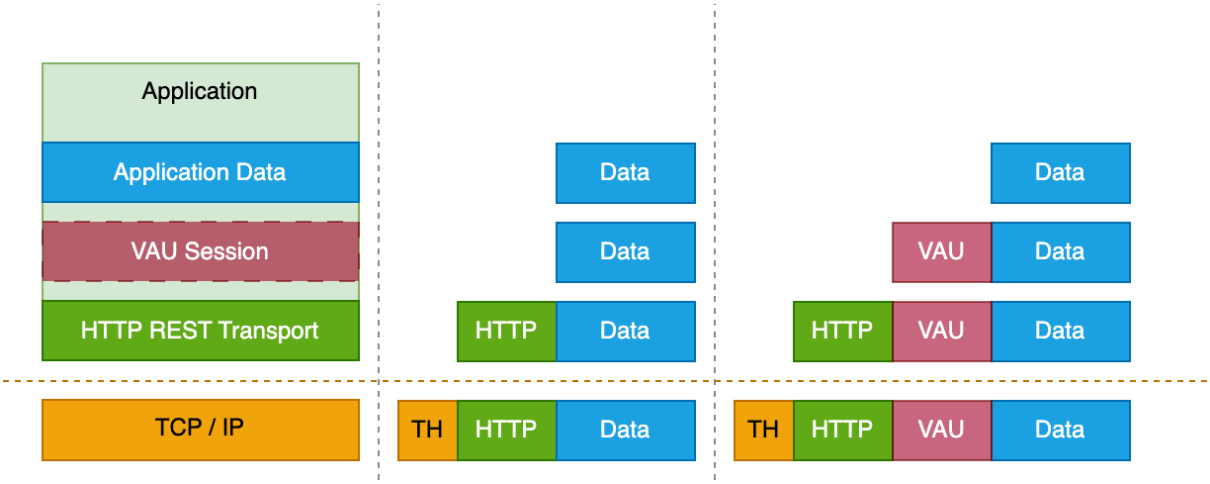
Within B²ric²s, the core modules are referred to as a `brick`. Each `brick` is a small, reusable component that solves a common problem. A group of such components, which together can solve a larger problem, are simply called `bricks`.

This documentation follows this pattern and presents each group of `bricks` individually.

RESTful-Bricks

The Representational State Transfer (REST) is a software architectural style created to describe the design and guide the development of the architecture for the World Wide Web. An application that adheres to the constraints of the REST architecture can be informally described as RESTful, although this term is more commonly associated with the design of HTTP-based APIs and what are widely considered best practices.

With the increasing popularity of REST, there are a variety of options to choose from for your goto library. This variety leads to the problem that we are dealing with many library interfaces. This makes cross-team reusability almost impossible. To address this problem, we first need to separate the concerns into layers, strongly inspired by the ISO OSI reference model.



One approach to solving this problem might be to restrict ourselves to a single library (or at least a couple of) and enforce it across all teams.

The `restful-bricks` implement the other option by enforcing their own abstraction layer `rest-client-api-brick` for RESTful operations.


```
<div class="imageblock"><div class="content"><img class="plantuml"
src="https://plantuml.gematik.solutions/plantuml/png/U9oLL5rlcp0KVUTNMDdBzk2CL4gQDWXrhPPzQ5gbPjjBjGSNhbCl
naR6h9cs_FVPV0M2IHiG4k1xpZttt6i7cVx38VGG8ah99_JNl24qZY6IYcYelX7wmWIiREWbclXukcZzlf5SgalqQaPzSvKhkv8cpqvZy
EtTOM19oY41eJiyDS3tkodjCqvvlcOYgm3EjDSasdnGCXTnpKHzUo-yVeRvlCnGSCDMWnA4L-
GE7khc0JWprs98qiuBtbQTo3PCf4IH14KoIQKmjHV1I46aYLXpy8AcQKB6jSDZagQSHKGpAHgp-
W9XIcVBPSTZhOZ8Kgdq4PYfYW9qdMxqdQPWya0cSDRjiZ2sKXB14TZC9nUJyvE57yAADluc-
M07uzaqEfyVDEKgqqJfFE5UuVZexltg6yqvkbOiscGZ50IDbZ2igE-tsEJ14MV68lnW8t-QGFCl3Ci01PdkhRU8-
b4aPVl8nhKgylKRDFhnxhjLK-
s1re9Lt5qvpluFn_99uA4gcdtvdCg6nrbiZyQL3OS3wFgKQefneZbGRIZSrhY8iMBhHuqjfk_Qpgj6OFdnqnswAO3b6DYurWxOpm6DCNr
agYU_4zmqolAw6kLcATqzTAWp9kOGheiTuAWTAahQZ_Jpm1eqOD7drUtN5rlqAvFY1HRTKvAPiTyctAxNvt7R9GVHoIwvECBXt3ll2aHi
ZeV_nRmtj000" /></div></div>
```


REST Client API Brick

The `rest-client-api-brick` is a REST client that can be used to communicate with a REST API. It is based on the simple idea of having a minimalist and simple API for sending HTTP requests and receiving HTTP responses. It is designed to be easy to use and understand and extensible to support different use cases.

Unlike many other libraries, it does not attempt to be a full-featured, heavyweight HTTP client. Instead, the `rest-client-api-brick` focuses on

supporting the HTTP protocol as simply as possible, especially from a user perspective.

<img class="plantuml"
src="https://plantuml.gematik.solutions/plantuml/png/U9oLKT5EmZ0KVUzNDF45DCDfeY6m408QUU094r-
C3sMxWuQkdUsTk2Z_RhTrkCbuiCsIzfvpppcx3Gpc0YQ4KgQrsjClUw9q446eD4DmTqgNNC9pGHtH6z-x6pVASoMK7j6B-
zYtgq8EbKnc_gVW5xkbm9DKG08IMpfrW--tBOjla8fimwLn3PssCnRkdhJAP5GhnNwnn-JKOJYi74fj-4JGaWadtg4JT-
i0tDerFXVfG89hzn9cnsNAD4jeg99KIVll9Pbg2979ZG0oDSWqPeaWNDeyCGk1BX3JsLpmtr5TKIuvzlfqL85X2Jh8R3DyK7lPwzS5a56
lvAtWFGE3nxuL61l303cGltwF4ZdcHxyFq8OhwHfVgfl3ji0YMx3OaXiC2iAYB4qSOQsYtABh7E7rhSFBnJejktJjUgLDI3FcfQUzIHDj
pko8ENQhC_Zsl7QASwWBCxNJjE_p0md31WG0" />

This means also that encoding and decoding the payload is not part of the brick itself. For payloads with different payload formats such as FHIR, JSON, XML, and others, you will need to implement your own "application client brick" (or reuse an existing one) which handles the encoding and decoding such as the `fd-fhir-client-brick` specifically for RESTful FHIR APIs.

By being a simple HTTP client, we get a bunch of additional benefits essentially for free, such as interchangeability and ease of implementation.

Let's dive into the tutorial on Using a HttpBClient

Using a HttpBClient

This tutorial is quite short and easy to understand, which is the basic idea of the `rest-client-api-brick`.

to keep this focused on the API, the instantiation of a concrete HttpBClient implementation is shown in a separate tutorial.

Let's demonstrate the `rest-client-api-brick` by the example:

```
HttpBClient client = fromSomewhere();

HttpBRequest request = HttpBRequest.post()
  .urlPath("/v1/fd/pingpong")
  .headers(
      HttpHeader.accept(MediaType.ANY_TYPE),
      HttpHeader.forContentType(MediaType.PLAIN_TEXT_UTF_8))
  .withPayload("Hello World");

HttpBResponse response = client.send(request);
int status = response.statusCode();
String payload = response.bodyAsString();
String contentType = response.contentType();
```

That's basically it. This approach is very simple, but extremely powerful. Dive into the next tutorials to learn how to use it for more complex scenarios.

Configuring a HttpBClient implementation

When you read the previous tutorial, did you wonder where to put the server URL? And what about TLS or some standard headers? In this tutorial we will see how this can be achieved with the `rest-client-brick`. This brick provides two different implementations for the API.

<img class="plantuml"
src="https://plantuml.gematik.solutions/plantuml/png/U9oLaTrAn30GXU_p50FUT96kLL2MjiYoA-Y5Loi-

```
G4YdQzXqKfEKLRJlRlgdMrilJ2YaEJFVETEcrhqhl6K0t1XzX0z_0bXcABJX3ljtW4T9-
DIKnd0PXTV9oVLMAsrYEBl98xywfUumbVqV8MhsY22BKc61v4QSeI6AnfRDionLjPTa-uPPsmqNXtkZAye6KXurEs5JXzMgSsZP-
ERG45SzV8QJptCGh_nA_efKi_IY_nFs8AdaXXSWT55gyhEtnM1GE4vxXMnj7JUkAXIJvFFaN20yE5Tkjah-
VAfpa2HTi82u4qKhzf9zgToTFbAm62wGig2js-5hXTPzz-tG-
XWMMSsOKDnQFxQLej46lxeNdaaQtpgMs6cqz3CCV_TF2oOSjlRn_C1V3GpC_G00" /></div></div>
```

The `BasicHttpClient` uses only the standard library, mainly `java.net`, to implement the interface. On the other hand, the `UnirestHttpClient` wraps Unirest. Both have their strengths and weaknesses, and you can use one or the other depending on your needs.

Regardless of the implementation, the API aims to achieve a separate client per server and user. This means that each client can be configured separately. The equivalent in `Unirest` is the `UnirestInstance`. This approach will come in handy once you start communicating with multiple applications or to a single application with multiple clients.

Let's see how to configure and instantiate the two implementations:

```
HttpBClient client = BasicHttpClient.forUrl("https://gematik-ti.de") // (1)
  .header( // (2)
      HttpHeader.acceptCharsetUtf8(),
      HttpHeader.accept(MediaType.ANY_TYPE)
  )
  .header( // (3)
      BasicHeaderProvider.forXRequestId(),
      BasicHeaderProvider.forDate()
  ).withoutTlsVerification(); // (4)
```

1. Initiate a Builder with the URL of the server you want to talk to
2. Set automatic headers which the client should set on each request
3. Set automatic header provides which generate http headers dynamically for each request
4. And finally instantiate the `HttpBClient` without TLS Verification

Dynamic header provider in step 3 will be explained in the Plugins tutorial in more detail.

Pretty simple, right? And by simply replacing the initial Builder call, you can easily replace your implementation: `HttpBClient client = UnirestHttpClient.forUrl("https://gematik-ti.de")`

But what if you want to use TLS verification? Both implementations have you covered here:

```
val builder = BasicHttpClient.forUrl("https://gematik-ti.de");

SSLContext sslCtx = SSLContext.getInstance("TLS");
TrustManager trustManager = new MyTrustManager(); // implement yourself

HttpBClient client1 = builder.withTlsVerification(sslContext);
HttpBClient client1 = builder.withTlsVerification(trustManager);
```

You can use either a `javax.net.ssl.SSLContext` SSLContext or a `javax.net.ssl.X509TrustManager` TrustManager. Both come from the standard

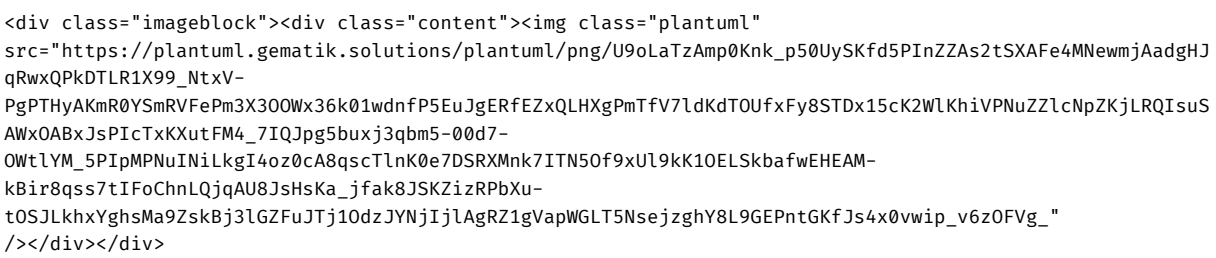
library and are well documented and known.

Give it a try and implement your own insecure TrustManager. That is basically exactly what the pre implemented clients do on `.withoutTlsVerification()`.

Notice how we created two separate clients (`assertNotEquals(client1, client2)`) in the previous example? This is exactly how it is supposed to work. A concrete client is instantiated on the "terminating builder method" and can be repeated as often as you like. You can use this approach to configure once and instantiate many times.

Using a HttpB Plugins

The `rest-client-api-brick` defines two different types of plugins following the Interface Segregation Principle.



```
<div class="imageblock"><div class="content"><img class="plantuml"
src="https://plantuml.gematik.solutions/plantuml/png/U9oLaTzAmp0Knk_p50UySKfd5PInZZAs2tSXAFe4MNewmjAadgHJ
qRwxQPkDTLR1X99_NtxV-
PgPTHyAKmR0YSmRVFePm3X3OOWx36k01wdnfP5EuJgERfEZxQLHXgPmTfV7ldKdTOUfxFy8STDx15cK2WlKhiVPNuZZlcNpZKjLRQIsuS
AWxOABxJsPIcTxKXutFM4_7IQJpg5buxj3qbm5-00d7-
OWtlYM_5PIpMPNuINiLkgI4oz0cA8qscTlnK0e7DSRXMnk7ITN5Of9xUl9kK1OELSkbafwEHEAM-
kBir8qss7tIFoChnLQjqAU8JsHsKa_jfak8JSKZizRPbXu-
tOSJLkhxYghsMa9ZskBj3lGZFuJTj1OdzJYNjIjlAgRZ1gVapWGLT5NsejzghY8L9GEPntGKfJs4x0vwip_v6zOFVg_"
/></div></div>
```

The `RequestHeaderProvider` is pretty much self-explanatory, it generates per each single request a dynamic `HttpHeader`. To demonstrate this kind of plugin, let's implement a simple `XRequestIdProvider`:

```
public class XRequestIdProvider implements RequestHeaderProvider {
  @Override
  public HttpHeader forRequest(HttpBRequest request) {
    val xReqId = UUID.randomUUID().toString();
    return new HttpHeader("x-request-id", xReqId);
  }
}
```

A slightly more complex example of a useful `RequestHeaderProvider` is the `smartcard-idp-plugin-brick` which can provide dynamic Header for the JWT Bearer Token from the smartcard IDP.

The other type of plugin is the `HttpBObserver`, which is heavily inspired by the Observer Pattern. This type of plugin is particularly useful when it comes to analysing or reporting on http traffic. To demonstrate this kind of plugin, let's implement a simple `HttpRecorder`:

```
public class HttpRecorder implements HttpBObserver {
  private final RawHttpCodec httpCodec = RawHttpCodec.defaultCodec();
  private final PrintStream out = System.out;

  @Override
  public void onRequest(HttpBRequest httpBRequest) {
    val request = httpCodec.encode(httpBRequest);
    out.println("--------- REQUEST ---------");
    out.println(request);
    out.println("--------- /REQUEST ---------\n");
```

```
    }

    @Override
    public void onResponse(HttpBResponse httpBResponse) {
      val response = httpCodec.encode(httpBResponse);
      out.println("--------- RESPONSE ---------");
      out.println(response);
      out.println("--------- /RESPONSE ---------\n");
    }
  }
```

Don’t worry, the `RawHttpCodec` will be explained later. For the sake of understanding here, what the `RawHttpCodec` does is transform a `HttpBRequest` and `HttpBResponse` into their string encoded form according to RFC standards.

FHIR-Bricks

While FHIR (Fast Health Interoperability Resources) is a powerful standard for exchanging healthcare data, it is also complex and can be difficult to understand. Luckily, there are some tools and libraries available that can help you work with the FHIR standard.

For the JVM (Java Virtual Machine), HAPI is the only open source implementation of the full standard. In fact, you could call HAPI the reference implementation. As such, it is the solution of choice for many developers in the healthcare industry worldwide.

However, even with HAPI, it can be challenging to use FHIR in a maintainable, reusable, and scalable way. To help with this, we have created a set of FHIR-Bricks, which are intended to be small, reusable, and extendable. Each FHIR-Brick solves a common problem that may arise when working with the FHIR standard.

<img class="plantuml"
src="https://plantuml.gematik.solutions/plantuml/png/U9oLLCjksp0GlEih2FVI7EHAGPgciI04JUksG4xjCSY1afOEOOeK
bcHUXV-zf1wsvCYA8qA0HCxExCwIJBHv5f06X514zKZ-kIz2vWNa2gc1zf-G6oxXhuTUaZWAFozwqzTAABma7yvPv9vcPTFGMFq-
XiYF0GClAm4bI3FWwGAYQ2Zfttabx8fBtGQCodwZ-
VedAYkBZeb5VYo2rmeN5ur2pGrF1b1IqPAFyB1n7e1JzomEfRG9aazj9_IQoueYBKckoaf9LtiD9WYveN8b84YsJPEgw1oUkuxjt1wja0
4ClTeg7Cbn7j6lyPSzlnr9gUH1H-
ACnd2wsA4buwk3wENoU_nZtv8hRIWQMuhWumcP8J10a3ayKC4BQXI6WcSpaYHTTz8qk7qAsJt7C4EUh_NTS36eRPijV_t-
GwvTnJxMLvwcRLwtTQpDRTaGtDK0NsQQzW05X1dLN5Ue61SXdasYCyj50TZ7UCaUufvMF6JsvONBrGIiiT1YNNeFvcVtW8Ldqi_QG3b9w
Cxxn7fhD5Vo5UWjjJ1C9xDsDyGRKkzb61Ip2nwhuSZG0pOUb_h4TkqksQsDUtjsa46_GOto9hW2MRXJyH-mFyHp" />

FHIR Coding System Brick

The FHIR standard is a set of rules and specifications for the exchange of health care data. It is designed to be flexible and adaptable, so that it can be used in a wide range of settings and with different health care information systems. Unfortunately, this flexibility can also make it difficult to work with HAPI. This comes primarily from the "being a general-purpose-library" nature of HAPI. Conversely, however, this also means that a consistent and maintainable usage across different projects and teams can be very challenging.

The `fhir-coding-system-brick` introduce an additional layer of abstraction on top of HAPI, which makes it easier to work with FHIR resources in a consistent and maintainable way. The `fhir-coding-system-brick` is a set of classes and interfaces that provide a more structured and type-safe way to work with elements of any FHIR resources. It is designed to be easy to use, and to provide a clear and consistent API that can be used across different projects and teams.

Wondering why you should use the `fhir-coding-system-brick`? Here are some reasons for and against using it.

Use it if:

- You are a complete beginner with FHIR. You want to get started quickly.
- You are tired of constantly having to deal with system urls of the elements and structures.
- You would like to have a more structured and type-safe way of working with FHIR resources.
- You are working on a project that requires interoperability with other projects

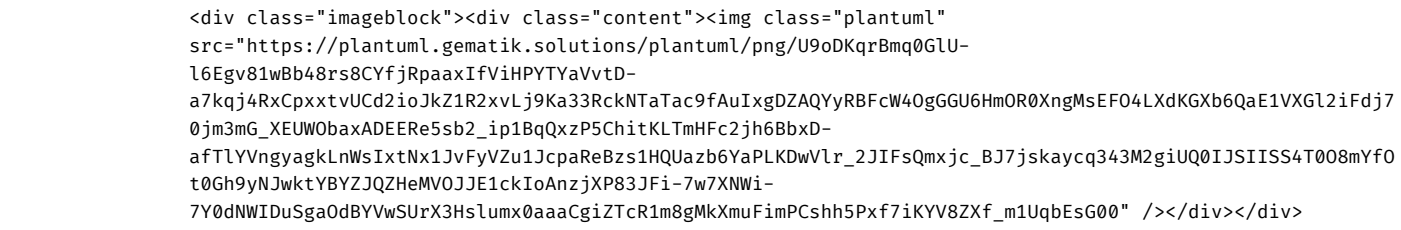
and teams.

Don’t use it when:

- You are working on a small project that doesn’t require a lot of complex FHIR resources.

Architecture

The `fhir-coding-system-brick` provides basically a set interfaces that represent the different elements and structures of a FHIR resource. These interfaces are designed to be easy to use, and to provide a clear and consistent API that can be used across different projects and teams:



The base FHIR specification describes only a set of base resources, while profiling FHIR allows to tailor the standard to different jurisdictions and domains and published as packages. Let’s take for example the Basisprofile DE (R4) package and see how the `fhir-coding-system-brick` can be used to implement it.

Implementing a FHIR package

In fact B²ric²s comes already with a basic implementation of the Basisprofil DE (R4) package, which is called `fhir-de-basisprofil-r4-brick`. We will build the `fhir-de-basisprofil-r4-brick` to demonstrate how to implement a FHIR package with the `fhir-coding-system-brick` from scratch.

Let’s dive in with a simple task!

Task: create the following Extension for *Normgröße* using plain HAPI FHIR Structures R4.

```
<extension url="http://fhir.de/StructureDefinition/normgroesse">
  <valueCode value="N1"/>
</extension>
```

```
var code = new CodeType("N4");
var ext  = new Extension("https://fhir.de/StructureDefinition/Normgroesse", code);
```

Pretty simple, right?

But what kind of code ist that anyway? Surely this thing "N4" must have some kind of meaning. And indeed it does. Basically, the URL `fhir.de/StructureDefinition/normgroesse` tells us exactly where we can find the definition of this object. But what if we would mix up the URL with the CodeSystem?

Have you spotted the issues in this small code snippet? Don’t

worry, the compiler wouldn't either, but a validator would notice that immediately.

Image this scenario in a larger project with multiple developers. Constant interruption and confusion about the system urls of the elements and structures slow you down, make you less productive and can lead to errors all the time. Just trust me for a second, you don't want to go there.

What if we think about an architectural approach to enforce the natural structure of the FHIR standard and to avoid those kind of issues. So, instead of passing all relevant data (something like `N1`) and meta-data (something like `fhir.de/CodeSystem/normgroesse`) around in a unstructured way, you could use the `fhir-coding-system-brick` to represent your information in an intuitive and supportive way.

Have you noticed again? Did we just mix up the definitions `fhir.de/StructureDefinition/normgroesse` and `fhir.de/CodeSystem/normgroesse`?

So basically, in order to implement the Basisprofil DE (R4) with `fhir-coding-system-brick` we need to implement the interfaces one by one.

Don't worry about gathering all the information from the Basisprofil DE (R4) package at once. You can always add more elements and structures later as you move on.

Step 1: implement a `DeBasisProfilNamingSystem` which will hold all the NamingSystems of the Basisprofil DE (R4) package.

```
@Getter
@RequiredArgsConstructor
public enum DeBasisProfilNamingSystem implements WithNamingSystem {
  IKNR("http://fhir.de/sid/arge-ik/iknr"),
  KVID_GKV("http://fhir.de/sid/gkv/kvid-10"),
  KVID_PKV("http://fhir.de/sid/pkv/kvid-10"),
  TELEMATIK_ID("https://gematik.de/fhir/sid/telematik-id");

  private final String canonicalUrl;
}
```

Remember, no need to have all the NamingSystems at once. When you discover a new NamingSystem, just add it to the enumeration.

Step 2: implement a `DeBasisProfilCodeSystem` which will hold all the CodeSystems of the Basisprofil DE (R4) package.

```
@Getter
@AllArgsConstructor
public enum DeBasisProfilCodeSystem implements WithCodeSystem {
  LAENDERKENNZEICHEN("http://fhir.de/CodeSystem/deuev/anlage-8-laenderkennzeichen"),
  VERSICHERUNGSART_DE_BASIS("http://fhir.de/CodeSystem/versicherungsart-de-basis"),
  IDENTIFIER_TYPE_DE_BASIS("http://fhir.de/CodeSystem/identifier-type-de-basis"),
  NORMGROESSE("http://fhir.de/CodeSystem/normgroesse"),
  PZN("http://fhir.de/CodeSystem/ifa/pzn");
```

```
  private final String canonicalUrl;
}
```

Step 3: implement a `Normgroesse` which will hold all the defined values of the Normgröße-ValueSet:

```
@Getter
@RequiredArgsConstructor
public enum Normgroesse implements FromValueSet {
  KA("KA", "Kein Angabe"),
  KTP("KTP", "Keine therapiegerechte Packungsgröße"),
  N1("N1", "Normgröße 1"),
  N2("N2", "Normgröße 2"),
  N3("N3", "Normgröße 3"),
  NB("NB", "Nicht betroffen"),
  SONSTIGES("Sonstiges", "Sonstiges");

  private final String code;
  private final String display;

  @Override
  public DeBasisProfilCodeSystem getCodeSystem() {
    return DeBasisProfilCodeSystem.NORMGROESSE;
  }

  public Extension asExtension() {
    return DeBasisProfilStructDef.NORMGROESSE.asCodeExtension(this.getCode());
  }
}
```

Step 4: You may have noticed that those FHIR packages can also evolve and change over time and thus must be versioned accordingly. To handle this, we implement the `DeBasisProfilVersion` which will hold all the versions of the Basisprofil DE (R4) package.

```
@Getter
@RequiredArgsConstructor
public enum DeBasisProfilVersion implements ProfileVersion {
  V1_3_2("1.3.2"),
  V1_4_0("1.4.0");

  private final String version;
  private final String name = "de.basisprofil.r4";
}
```

Step 5: Once we have the versions we want to deal with, we can finish the profile definition by implementing the `DeBasisProfilStructureDefinition` which will hold all the StructureDefinitions of the Basisprofil DE (R4) package.

```
@Getter
@RequiredArgsConstructor
public enum DeBasisProfilStructDef implements WithStructureDefinition<DeBasisProfilVersion> {
  GKV_VERSICHERTENART("http://fhir.de/StructureDefinition/gkv/versichertenart"),
  NORMGROESSE("http://fhir.de/StructureDefinition/normgroesse"),
  HUMAN_NAMENSZUSATZ("http://fhir.de/StructureDefinition/humanname-namenszusatz");

  private final String canonicalUrl;
}
```

You've done it!

OK, so this approach requires a bit more code to set up. But hang on a sec.

We have now a solid way of working with the elements and structures of the Basisprofil DE (R4) package that is structured and type-safe.

Advantages are:

1. Structured and logical interface for dealing with the FHIR coding system
2. Many common use cases for creating FHIR elements are already implemented for you
3. No more fiddling with those awkward system URLs.
4. Implementations of `fhir-coding-system-brick` are standardized and can be reused across different projects and teams.
5. Fits naturally into the `fhir-bricks` ecosystem.

Disadvantages are:

1. tediously setting up the initial structure

To demonstrate the advantages of the `fhir-coding-system-brick`, let's return to the original task.

```
<extension url="http://fhir.de/StructureDefinition/normgroesse">
  <valueCode value="N1"/>
</extension>
```

It's really just the value code that we already have in place as an enumeration called `Normgroesse`.

```
Extension ext = Normgroesse.N1.asExtension();
```

That's all it takes, and it reads like a charm. And it gets even better, `fhir-coding-system-brick` gives your code some superpowers. Remember how we have implemented `Normgroesse.N1.asExtension()`:

```
public Extension asExtension() {
  return DeBasisProfilStructDef
          .NORMGROESSE       // StructureDefinition/normgroesse
          .asCodeExtension(  // create as an extension for a value code
              this.getCode() // this == Normgroesse.N1
          );
}
```

In addition to extensions, it is also very common that codes or value sets have to be represented as a `CodeableConcept`. The `fhir-coding-system-brick` already has default implementations for many of the common use cases. For example, it is directly possible to represent the `Normgroesse` as a `CodeableConcept` without having to explicitly implement any additional method for this:

```
CodeableConcept cc = Normgroesse.NB.asCodeableConcept();
```

Which will encode in XML to:

```
<code>
  <coding>
     <system value="http://fhir.de/CodeSystem/normgroesse"/>
     <code value="KTP"/>
  </coding>
</code>
```

Implementing a custom FHIR Resource

When working with native HAPI resources, you often have to deal with extracting certain values. Due to the 'general purpose' nature of HAPI, you need to extract your objects from the FHIR data model.

The call looks something like this:

```
Bundle bundle = getBundleFromSomewhere();
Medication medication = bundle.getEntry().stream()
    .filter(entry -> entry.getResource().getResourceType().equals(ResourceType.Medication))
    .filter(KbvStructDef.KBV_MEDICATION_PZN::matches)
    .map(entry -> (Medication)entry.getResource())
    .findFirst()
    .orElseThrow();
```

This is not only extremely time-consuming, but also difficult to maintain. This is where custom resources come into play. Let's take a look at how a 'KbvErpBundle' can make our lives a lot easier:

```
public class KbvErpBundle extends Bundle {
    public Medication getErpMedication() {
      return this.getEntry().stream()
          .filter(entry -> entry.getResource().getResourceType().equals(ResourceType.Medication))
          .filter(KbvStructDef.KBV_MEDICATION_PZN::matches)
          .map(entry -> (Medication)entry.getResource())
          .findFirst()
          .orElseThrow();
    }
}
```

So now whenever we have a `KbvErpBundle` object, and we need to extract the medication object, we can just do that:

```
KbvErpBundle bundle = getKbvErpBundleFromSomewhere();
Medication medication = bundle.getErpMedication();
```

And we can implement this concept with all the other FHIR resources you need to work with. And we can implement this concept with all the other FHIR resources you need to work with. Let's have a look at what a `KbvErpMedication` might look like.

```
private class KbvErpMedication extends Medication {
  public StandardSize getStandardSize() {
    return this.getExtension().stream()
        .filter(DeBasisProfilStructDef.NORMGROESSE::matches)
        .map(ext -> StandardSize.fromCode(ext.getValue().castToCoding(ext.getValue()).getCode()))
        .findFirst()
        .orElse(StandardSize.KA);
  }
}
```

```
public class KbvErpBundle extends Bundle {
    public KbvErpMedication getErpMedication() {
      return this.getEntry().stream()
          .filter(entry -> entry.getResource().getResourceType().equals(ResourceType.Medication))
          .filter(KbvStructDef.KBV_MEDICATION_PZN::matches)
          .map(entry -> (KbvErpMedication)entry.getResource())
          .findFirst()
          .orElseThrow();
    }
}
```

And now we can really treat our own FHIR objects as plain old Java objects:

```
KbvErpBundle bundle = getKbvErpBundleFromSomewhere();
KbvErpMedication medication = bundle.getErpMedication();
StandardSize size = medication.getStandardSize();
```

Unfortunately, that is only half the story. The next tutorial will show you how to use a `TypeHint`.

Matching FHIR Resources and Elements

Have you noticed how we have implemented the `getStandardSize()` method in the KbvErpMedication from this tutorial? We have used a `filter` method to extract the `StandardSize` from the `Extension`-List of the `Medication`.

This is a very common pattern when dealing with FHIR. Imagine a FHIR resource containing a list of extensions. If you want to select a particular one of them, the profile (to be precise the canonical URL of the profile a resource claims to conform) is your buddy here.

If you have completed this tutorial you already have everything you need to match the systems of your package in. And once you start using custom resources you will quickly find that you need to filter your items out of HAPI lists quite often.

Let's have a closer look at the condensed example from the custom resources tutorial.

```
Bundle bundle = getBundleFromSomewhere();
Medication medication = bundle.getEntry().stream()
    .filter(entry -> entry.getResource().getResourceType().equals(ResourceType.Medication))
    .filter(KbvStructDef.KBV_MEDICATION_PZN::matches)
    .map(entry -> (Medication)entry.getResource())
    .findFirst()
```

```
    .orElseThrow();
```

To understand what's going on here, let's look at an example bundle in plain xml:

```
<Bundle xmlns="http://hl7.org/fhir">
  ...
  <entry>
    <resource>
      <Composition>
        <meta>
          <profile value="https://fhir.kbv.de/StructureDefinition/KBV_PR_ERP_Composition|1.1.0" />
        </meta>
        ...
      </Composition>
    </resource>
  </entry>
  <entry>
    <resource>
      <MedicationRequest>
        <meta>
          <profile value="https://fhir.kbv.de/StructureDefinition/KBV_PR_ERP_Prescription|1.1.0" />
        </meta>
        ...
      </MedicationRequest>
    </resource>
  </entry>
  <entry>
    <resource>
      <Medication>
        <meta>
          <profile value="https://fhir.kbv.de/StructureDefinition/KBV_PR_ERP_Medication_PZN|1.1.0"
/>
        </meta>
        ...
      </Medication>
    </resource>
  </entry>
  <entry>
    <resource>
      <Patient>
        <meta>
          <profile value="https://fhir.kbv.de/StructureDefinition/KBV_PR_FOR_Patient|1.1.0" />
        </meta>
        ...
      </Patient>
    </resource>
  </entry>
  <entry>
    <resource>
      <Practitioner>
        <meta>
          <profile value="https://fhir.kbv.de/StructureDefinition/KBV_PR_FOR_Practitioner|1.1.0" />
        </meta>
        ...
      </Practitioner>
    </resource>
  </entry>
  <entry>
    <resource>
      <Organization>
        <meta>
          <profile value="https://fhir.kbv.de/StructureDefinition/KBV_PR_FOR_Organization|1.1.0" />
        </meta>
        ...
      </Organization>
    </resource>
  </entry>
```

```
    <entry>
      <resource>
        <Coverage>
          <meta>
            <profile value="https://fhir.kbv.de/StructureDefinition/KBV_PR_FOR_Coverage|1.1.0" />
          </meta>
          ...
        </Coverage>
      </resource>
    </entry>
  </Bundle>
```

In general, we have at least two ways of finding a particular resource from the entries in a bundle:

1. we use the `ResourceType` to filter the entries
2. we use the `meta.profile` (system url) to filter the entries

In the original code, we used both approaches simultaneously, which is perfectly fine, but shouldn't be necessary in most cases. The `meta.profile` is the more specific way to filter the entries, as it is unique for each resource type. The `ResourceType` is more general and can be used to filter out all resources of a certain type. So when we apply the following filter, we know implicitly from the structure definition that this resource must be of the desired `ResourceType`:

```
Bundle bundle = getBundleFromSomewhere();
Medication medication = bundle.getEntry().stream()
    .filter(KbvStructDef.KBV_MEDICATION_PZN::matches)
    .map(entry -> (Medication)entry.getResource())
    .findFirst()
    .orElseThrow();
```

What the filter does is basically compare the given system url of a specific `WithSystem` object with the `meta.profile` of a resource.

Having such a powerful mechanism makes the custom resources tutorial easier to implement and test.

FHIR Builder Brick

The FHIR standard is a set of rules and specifications for the exchange of health care data. It is designed to be flexible and adaptable, so that it can be used in a wide range of settings and with different health care information systems. Unfortunately, this flexibility can also make it difficult to work with HAPI. This comes primarily from the "being a general-purpose-library" nature of HAPI. Conversely, however, this also means that a consistent and maintainable usage across different projects and teams can be very challenging.

To solve this issue, the `fhir-builder-brick` provides a simple concept based on the Builder Pattern. This allows you to build FHIR resources in a structured and type-safe way by applying the same principles we have already used for matching resources.

<img class="plantuml"
src="https://plantuml.gematik.solutions/plantuml/png/U9o5KbzhmZ0GV--
dEEQBCWUo5w4KaNPMX85WFq5CRrreUYcN5BQ9ttrNgqFTO7aATxz_kSlI1yMXQspaQqEjOjL0nQvhD_IER88YZV1yrMoLhbM5HVYq2BbK
CF8OWEJYfOS5zhZW6CWHHbTCxPfMIXJW486SL9HwAoepPnqd2Nu4P59siO1HbkNvQZRW79V8lp4lYkir8r9qZ29jbVUGgnevPQDhI18OP

zDHdi_diza42aELnU18Y8FzqriFZWVEgV881vb6CFe85OOrSgDeF87uIlOFtAlJofelBE-
XaahjVM2b0mpnKkKnxOoLvoIxAQH354RlEjQuAMCe0ajGAHukr2FiUyPu4iEkbxpNt9tPjxhzcAYySRmdhYmsiep_UH6ZbdyWuuE7p95c
30YcQGU-pC7HmsLXvzaE3jiM-TJUTa6MVn7yCRcBBgzRIbt-uZVOhE5K" />

The `fhir-builder-brick` provides basically a set of base classes which need to be extended and implemented for your own needs. The architecture will guide you through the process of building FHIR resources in a structured and flexible way.

Wondering why you should use the `fhir-builder-brick`? Here are some reasons for and against using it.

Use it if:

- You are new to FHIR. You want to get started quickly. The `fhir-bricks` will guide you on your journey.
- You are tired of constantly having to deal with the system urls of the elements and structures.
- You want to have a more structured and type-safe way to working with FHIR
- You are working on a project that requires interoperability with other projects and teams.

Use it optionally if:

- You have a small project that doesn't require a lot of complex FHIR resources.
- You don't have an implementation of `fhir-coding-system-brick`.

Don't use it if:

- You already have your own implementation of the Builder Pattern that is too complex to refactor.

ResourceBuilder

The `ResourceBuilder` is designed to simplify and standardise the process of creating FHIR Resources. This base class has the following signature:

```
public abstract class ResourceBuilder<R extends Resource, B extends ResourceBuilder<R, B>> {
  // we'll cover this later
  public abstract R build();
}
```

The `ResourceBuilder` does so by:

1. providing some common methods which are used in every builder
2. enforcing the `build()`-method, making the builders uniform and intuitively to use

But why those generics?

1. `R` defines the builder to be specific for this special kind of `Resource`
2. the builder MUST generate native HL7/HAPI resources or HAPI FHIR Custom Structures
3. `B` on the other hand tells the `ResourceBuilder` to return the specific builder instance on common functions

ElementBuilder

The `ResourceBuilder` is designed to simplify and standardize the building of FHIR Types.

```
public abstract class ElementBuilder<E extends Element, B extends ElementBuilder<E, B>> {
  // we'll cover this later
  public abstract R build();
}
```

The `ElementBuilder` does so by:

1. providing some common methods which are used in every builder
2. enforcing the `build()`-method, making the builders uniform and intuitively to use

But why those generics?

1. `E` defines the builder to be specific for this special kind of `Element`
2. the builder MUST generate native HL7/HAPI resources or HAPI FHIR Custom Structures
3. `B` on the other hand tells the `ElementBuilder` to return the specific builder instance on common functions

FakerBrick

When dealing with test data, it is often necessary to generate random or fake data. The `FakerBrick` provides a set of classes and interfaces that allow you to generate fake data for FHIR resources in a structured and extensible way.

Implementing a FHIR Resource Builder

Now that we have laid the foundations in the last tutorial, we can start building entire FHIR resources.

Task: create the following `Medication` resource using plain HAPI FHIR Structures R4.

```
<Medication xmlns="http://hl7.org/fhir">
  <id value="47076fb4-dc5c-4f75-85f6-b200033b3280" />
  <meta>
    <profile value="https://fhir.kbv.de/StructureDefinition/KBV_PR_ERP_Medication_PZN|1.1.0" />
  </meta>
  <extension url="http://fhir.de/StructureDefinition/normgroesse">
    <valueCode value="N1" />
  </extension>
  <code>
    <coding>
      <system value="http://fhir.de/CodeSystem/ifa/pzn" />
      <code value="03879429" />
    </coding>
    <text value="Beloc-Zok® mite 47,5 mg, 30 Retardtabletten N1" />
  </code>
</Medication>
```

```
var medication = new Medication();
```

```
medication.getMeta().addProfile("https://fhir.kbv.de/StructureDefinition/KBV_PR_ERP_Medication_PZN|1
.1.0");
medication.addExtension("http://fhir.de/StructureDefinition/normgroesse", new CodeType("N1"));
medication.setCode(new CodeableConcept().addCoding(new Coding("http://fhir.de/CodeSystem/ifa/pzn",
"03879429", "Beloc-Zok® mite 47,5 mg, 30 Retardtabletten N1")));
```

It's getting harder, isn't it? Now we must track down a whole bunch of different systems coming from different packages in different versions.

Warning Have you spotted the issues in this small code snippet? Don't worry, the compiler wouldn't either, but a validator would notice that immediately.

For such a scenario the `fhir-builder-brick` comes in handy. It provides a set of classes and interfaces that allow you to build FHIR resources in a structured and type-safe way. It is designed to be easy to use, and to provide a clear and consistent API that can be used across different projects and teams.

For this task, we have to introduce a new package called KBV E-Rezept. We assume the coding being already implemented according to this tutorial. Having those in place you could already simplify the above code to:

```
var medication = new Medication();
medication.getMeta().addProfile(KbvItaErpStructDef.MEDICATION_PZN.getVersionedUrl(KbvItaErpVersion.V
1_1_0));
medication.addExtension(StandardSize.N1.asExtension());
medication.setCode(PZN.from("03879429").asNamedCodeable("Beloc-Zok® mite 47,5 mg, 30 Retardtabletten
N1"));
```

But it gets even better!

The `fhir-builder-brick` provides a simple interface to apply the Builder Pattern, which significantly improves the readability of your code.

Step 1: implement the basic structure for a `MedicationBuilder` class:

```
class MedicationBuilder extends ResourceBuilder<Medication, MedicationBuilder> {

    @Override
    public Medication build() {
      return null;
    }
}
```

Recap to ResourceBuilder to learn more about those generics.

Step 2: implement the `build`-Method with the code from the first or second example:

```
class MedicationBuilder extends ResourceBuilder<Medication, MedicationBuilder> {

    @Override
    public Medication build() {
      var medication = this.createResource(Medication::new, KbvItaErpStructDef.BUNDLE,
KbvItaErpVersion.V1_1_0);
```

```
        medication.addExtension(StandardSize.N1.asExtension());
        medication.setCode(PZN.from("03879429").asNamedCodeable("Beloc-Zok® mite 47,5 mg, 30
Retardtabletten N1"));
        return medication;
    }
}
```

Here the `ResourceBuilder` supports us with the `createResource`-method, which is a simple factory method to create a new resource instance and set the profile automatically. From the user perspective we already a nice and clean API to build a `Medication` resource:

```
Medication medication = new MedicationBuilder().build();
```

But what about those hardcoded values like the PZN or the name of the medication? Do they stay fixed or can we do better and provide some sort of setters? That's a perfect use case for the Builder Pattern.

Step 3: implement the `MedicationBuilder` with the Builder Pattern:

```
class MedicationBuilder extends ResourceBuilder<Medication, MedicationBuilder> {

    // default values are optional but sometimes pretty handy
    private KbvItaErpVersion version = KbvItaErpVersion.V1_1_0;
    private StandardSize size = StandardSize.N1;
    private CodeableConcept medicationCode;

    public static MedicationBuilder builder() {
      return new MedicationBuilder();
    }

    public MedicationBuilder withVersion(KbvItaErpVersion version) {
      this.version = version;
      return this;
    }

    public MedicationBuilder withStandardSize(StandardSize size) {
      this.size = size;
      return this;
    }

    public MedicationBuilder withMedication(String pzn, String name) {
      this.medicationCode = PZN.from(pzn).asNamedCodeable(name);
      return this;
    }

    public MedicationBuilder withMedication(PZN pzn, String name) {
      this.medicationCode = pzn.asNamedCodeable(name);
      return this;
    }

    @Override
    public Medication build() {
      var medication = this.createResource(Medication::new, KbvItaErpStructDef.BUNDLE, version);
      medication.addExtension(size.asExtension());
      medication.setCode(medicationCode);
      return medication;
    }
}
```

And from the user perspective:

```
var medication = MedicationBuilder.builder()
    .setId("47076fb4-dc5c-4f75-85f6-b200033b3280")
    .withVersion(KbvItaErpVersion.V1_0_2)
    .withStandardSize(StandardSize.N3)
    .withMedication("03879429", "Beloc-Zok® mite 47,5 mg, 30 Retardtabletten N1")
    .build();
```

Much better, isn't it? And because we are using `this.createResource` in our `MedicationBuilder` we could even leave out setting the `id` because a random UUID will be generated automatically. In fact, you never need to specify it explicitly unless you really need a specific one.

```
var medication = MedicationBuilder.builder()
    .withStandardSize(StandardSize.N3)
    .withMedication("03879429", "Beloc-Zok® mite 47,5 mg, 30 Retardtabletten N1")
    .build();
```

Instantiating with `MedicationBuilder.builder()` is just a convention for the Builder Pattern. As we already provide the information (*being a builder*) in the class name. You can repurpose this method freely, or just use the default constructor.

FHIR Codec Brick

So far we have only covered the creation and handling of FHIR objects. We have deliberately left out the aspect of serialisation. This is where the `fhir-codec-brick` comes into action.

A codec is a component that encodes and decodes a data stream or signal. And a `FhirCodec` in the `fhir-bricks` ecosystem is a component that can encode and decode FHIR objects.

<img class="plantuml"
src="https://plantuml.gematik.solutions/plantuml/png/U9nraazhmp0CnU_-
53fkRBtiQ4Of50gzZRRixZbQwjMHWwoCbB7lFkKVMzhE9_7Upv8UzYgBOscgQF8fKEtOLL1oQkej7P63EF88Jt_CslcJAt4lvuYmKGLDH
W7I8gk7U-muIGoK28q99CZlQi2hYw7edCqnC7mPqFCvQ7Zdat8a5lR2WSfxE-4XqGvp4sNEC-
RKi4SBkx7wxqh8lNnZmbjA4HtDg9jz9_3R61zTpj0bMAS2_HZ3Isj7KOUqm-2sYY-
ECxA5xL3qyaTETAdtnWCWUMrwUuTXwn7KoHs8ROrUi3YSQuNNtMhFX-KZN4Szp2zU0zEaBkDlkWM7yYYBvSNRMUqJiTHtC6Q5LEZt-
G6ESyTM" />

In addition to a normal codec, the `FhirCodec` is able to validate the correctness of the FHIR objects by delegating a `ValidatorFhir`. This topic will be covered in the FHIR Validation Brick.

Instantiating a FHIR Codec

Where can we get one of these `FhirCodec`? Can we customise it and do we need a `fhir-coding-system-brick`?

Let's find out!

The simplest way to instantiate a simple `FhirCodec` is as follows:

```
FhirCodec fhirCodec = FhirCodec.forR4().andDummyValidator();
```

This will instantiate a `FhirCodec` for the R4 version of FHIR and add a dummy validator to it. The dummy validator will always return `true` for any resource that is passed to it.

It might be more appropriate to use a method name such as `andNoValidator()` or `withoutValidation()`. However, `andDummyValidator()` describes exactly its purpose: it will instantiate the `FhirCodec` with a `DummyValidator`.

So basically what happens under the hood:

```
FhirCodec fhirCodec = FhirCodec.forR4().andCustomValidator(new DummyValidator(FhirContext.forR4()));
```

Slightly more code but the same result. Now we can look at the `andCustomValidator` method in more detail. As the name suggests, we can pass anything to this method that is a `ValidatorFhir`. Validation of FHIR resources is explained in more detail in the FHIR Validation Brick.

Decoding with Type Hints

The following challenge arises when a decoder attempts to convert a raw string into a structured object. The `FhirCodec` also faces the same challenge. Try to put yourself in the decoder's shoes to mentally understand the process.

Given the following XML snippet, how would you decide on which Java type to use for this object?

```
<Medication xmlns="http://hl7.org/fhir">
  <id value="47076fb4-dc5c-4f75-85f6-b200033b3280" />
  <meta>
    <profile value="https://fhir.kbv.de/StructureDefinition/KBV_PR_ERP_Medication_PZN|1.1.0" />
  </meta>
  <extension url="http://fhir.de/StructureDefinition/normgroesse">
    <valueCode value="N1" />
  </extension>
  <code>
    <coding>
      <system value="http://fhir.de/CodeSystem/ifa/pzn" />
      <code value="03879429" />
    </coding>
    <text value="Beloc-Zok® mite 47,5 mg, 30 Retardtabletten N1" />
  </code>
</Medication>
```

Roughly speaking, only the first line is enough. From this, the HAPI can directly deduce that it is decoding this object as `org.hl7.fhir.r4.model.Medication`. But the HAPI can do more. Did you notice that this `<Medication>` object contains a structure definition? The HAPI can analyse this profile and even offers the possibility to register custom structures.

```
FhirCodec codec = getFhirCodecFromSomewhere();
String content = readXmlExample();
```

```
// options to decode the Medication
Resource medication1 = fhirCodec.decode(content);
Resource medication2 = fhirCodec.decode(null, content);
Medication medication3 = fhirCodec.decode(Medication.class, content);
KbvErpMedication medication4 = fhirCodec.decode(KbvErpMedication.class, content);

// where these will fail with a ClassCastException
Patient patient = fhirCodec.decode(Patient.class, content);
```

These are the options we have, with the method to `medication1` being just a shortcut for `medication2`. Things get more exciting when you try to decode the others. Do you remember the first decoder challenge? The decoder overcomes this problem with your help. You literally tell the codec in the first argument what type of object to decode.

Consequently, an `ClassCastException` will be thrown when you instruct the codec to decode the `<Medication>` as `org.hl7.fhir.r4.model.Patient`.

```
// where these will fail with a ClassCastException
Patient patient = fhirCodec.decode(Patient.class, content);
```

But what happens to composite FHIR resources? For example, when we decode a `<Bundle>`, we can specify the concrete type `KbvErpBundle`, but what happens to the resources it contains? Basically the same as with any other object. The decoder has to decode every single resource. This is where the TypeHints come in.

Imagine having the following (simplified) `<Bundle>` which want to decode and work with:

```
<Bundle xmlns="http://hl7.org/fhir">
  <id value="1f339db0-9e55-4946-9dfa-f1b30953be9b" />
  <meta>
    <lastUpdated value="2022-05-20T08:30:00Z" />
    <profile value="https://fhir.kbv.de/StructureDefinition/KBV_PR_ERP_Bundle|1.1.0" />
  </meta>
  <identifier>
    <system value="https://gematik.de/fhir/erp/NamingSystem/GEM_ERP_NS_PrescriptionId" />
    <value value="160.100.000.000.037.28" />
  </identifier>
  <type value="document" />
  <timestamp value="2022-05-20T08:30:00Z" />
  <entry>
    <fullUrl value="http://pvs.praxis.local/fhir/Medication/5ff1bd22-ce14-484e-be56-d2ba4adeac31" />
    <resource>
      <Medication xmlns="http://hl7.org/fhir">
        <id value="5ff1bd22-ce14-484e-be56-d2ba4adeac31" />
        <meta>
          <profile value="https://fhir.kbv.de/StructureDefinition/KBV_PR_ERP_Medication_PZN|1.1.0"
/>
        </meta>
        <extension url="http://fhir.de/StructureDefinition/normgroesse">
          <valueCode value="N1" />
        </extension>
        <code>
          <coding>
            <system value="http://fhir.de/CodeSystem/ifa/pzn" />
            <code value="03879429" />
          </coding>
          <text value="Beloc-Zok® mite 47,5 mg, 30 Retardtabletten N1" />
        </code>
      </Medication>
```

```
    </resource>
  </entry>
</Bundle>
```

To decode this bundle, we can use the `KbvErpBundle` we created in Implementing a custom FHIR Resource.

```
String content = readXmlExample();

KbvErpBundle prescription = fhirCodec.decode(KbvErpBundle.class, content);
KbvErpMedication medication = prescription.getErpMedication(); // throws ClassCastException
```

However, when we call the custom method `getErpMedication()` we run into a `ClassCastException`. Let me explain why this is the case. When calling the `decode` method, we can only tell the decoder to decode the outer FHIR resource as a `KbvErpBundle`. But you cannot tell the decoder directly how to decode the `<Medication>`. This is where the `TypeHint` comes in.

A `TypeHint` allows you to tell the decoder in advance how to decode certain resources. This is basically the main purpose of the structure definition in the profile values.

```
<profile value="https://fhir.kbv.de/StructureDefinition/KBV_PR_ERP_Bundle|1.1.0" />

<profile value="https://fhir.kbv.de/StructureDefinition/KBV_PR_ERP_Medication_PZN|1.1.0" />
```

With a `TypeHint` we can tell the decoder that whenever it encounters this profile, it will decode it as `KbvErpMedication`. A `TypeHint` is given to the codec in advance. So we need to slightly modify the instantiation of our `FhirCodec`.

```
String content = readXmlExample();

FhirCodec fhirCodec = FhirCodec.forR4()
          .withTypeHint(KbvStructDef.KBV_MEDICATION_PZN, KbvErpMedication.class)
          .andDummyValidator();

KbvErpBundle prescription = fhirCodec.decode(KbvErpBundle.class, content);
KbvErpMedication medication = prescription.getErpMedication(); // no more exceptions
```

FHIR Configurator Brick

The FHIR packages that we want to map with the HAPI are subject to the same development cycle as classic software. Bug fixes and refactorings are carried out, but new features are also added or old ones are dropped. This can lead to the point where you need to handle different versions of the same package. Remember when we sneakily introduced the `KbvItaErpVersion` in the ResourceBuilder Tutorial?

As this undertaking can be very tricky, the `fhir-configurator-brick` provides a structure to handle different versions of FHIR packages. It is designed to be easy to use and understand.

<img class="plantuml"
src="https://plantuml.gematik.solutions/plantuml/png/U9nrKi5gmp0ClUSh1Bkqsmfbbq8feP0I6Emm2EpkkMegwbZ1LWQb

zD-dj2jTCiyd-xqdloVPwoWcIDUuB1x8joQO1kh0NVlgznX8ZBS8Bx_8rjY3gR6IeqCe5S4iee3NJLGEA-
nrmW4yUymowqoCy1vuHmvZmNv7TNSLh5OmAPuVodAncC-
dK96l7GhxF8TJ1hgUuAJnXEmPQfHxBR6Vz8QmX4g2barrbx98tBB1dUdSwB9_wy-Z1YeKKR-u4Vx9IDjRYClHi-
00Q7Wx1AZs7B1Gk3f6mIOkuOsYhAwI_AB143YKv0J3Y1q7kWVf1p8m-
i8GjRa1PhbfjULFXnzNDkLkZJuTMUCAPoYPZnkIsxp6j2PCllRZB52zbrcUc6MM6EyVVQzRez_gP_q6sYhlNW00" />

With this structure the `ProfilesConfigurator` can read your configuration which looks in yaml like this:

```yaml
- id: "1.4.0"
  note: "erp-workflow valid from 1.4.25"
  profiles:
    - name: "kbv.ita.erp"
      version: "1.1.2"
      compatibleVersions: "1.1.0"
      canonicalClaims: "https://fhir.kbv.de/"
    - name: "kbv.basis"
      version: "1.3.0"
      canonicalClaims: "https://fhir.kbv.de/"
      omitProfiles: [ "KBV_VS_Base_Diagnosis_SNOMED_CT.json", "KBV_VS_Base_Device_SNOMED_CT.json" ]
    - name: "de.gematik.erezept-workflow.r4"
      version: "1.4.0"
      compatibleVersions: [ "1.4" ]
      canonicalClaims: [ "https://gematik.de/fhir", "http://gematik.de/fhir" ]
  errorFilter:
    - "^2 profiles found for contained resource.*"
  ignoreCodeSystems:
    - "http://fhir.de/CodeSystem/ask"
    - "http://fhir.de/CodeSystem/ifa/pzn"
    - "http://fhir.de/CodeSystem/bfarm/atc"
```

FHIR Validation Brick

In addition to configuring, building and serializing, the FHIR standard also allows us to validate the correctness of resources.

Fortunately, the B²ric²s provides a simple and easy to use API for validating FHIR resources. The `fhir-validation-brick` provides a structured way to validate any FHIR resource. It is designed to be easy to use and to provide a clear and consistent API that can be used across projects and teams.

<img class="plantuml"
src="https://plantuml.gematik.solutions/plantuml/png/U9oLKx5Emp0KtFqLbbW0qGwC5QewG2M60ce42s8myIMnwZn7pqwX
8FuTXwI5g2rgCpdlxjxpkvCdFYWETMc5y0j3bM9LofnTNTrI0JP1KGfv-
GUiLBfGEPAmif3JM87m29BYmKSC2HfUS2p94OGm5C1P1EIJiaOtoBGmB3-5ZD-ohU4qTP57OIIJm8Roiz6QRXpDuMiR-
do6TpMd6CbvTzedCVwdl6F2gtCMYdgidNtNn2yXH6gLz_A1NMOiz6OdQShAeem3V7_HJZ2Brp57g-
uS7QsP8mE3FWuMNDTbkJg0BRhefSOmHnbzNWmPsRABicts6y7CwTgYyQndsSrxO9KsWrePslyEtP7IVTLqK_QghOoxk1wfwfn9e3Wj4eH
_8cwIsnF1-S0Y2uFnTgRx50FjtkXNiDb8R5i_U3vXw0kjV07zqkcsF18Ja8wFy1kEfqfc" />

In addition, the `fhir-validation-brick` allows you to implement your own validation strategies or use those already in place.

Strategies already implemented are:

1. `DummyValidator` is a dummy validator that always returns `true` (which can be seen as an option to disable the validation).
2. `NonProfiledValidator` is a validator that only knows the basic FHIR rules, but no specific profiles.
3. `ProfiledValidator` is a validator that additionally knows a specific set of profiles in a single configuration.

4. `MultiProfiledValidator` is a composition of multiple `ProfiledValidators`. This allows you to validate resources against multiple versions of a set of profiles.
5. Finally, the `ReferenzValidator` which is a wrapper around the Gematik Referenzvalidator

Disable FHIR Validation

There is no way to explicitly disable FHIR validation. However, by using the `DummyValidator` we can achieve pretty much the same result. Since `DummyValidator` always returns `true`, you can always take the happy path in your code. This helps you to avoid a lot of conditionals and gives you other advantages inherited from the Null Object Pattern.

Let's have a look at the following unit test to explore the capabilities of the `DummayValidator`

```
@Test
void shouldValidateSimpleString() {
    String content = "Hello Bbriccs!";
    ValidatorFhir validator = new DummyValidator(FhirContext.forR4());
    assertTrue(validator.isValid(content));
}
```

So here we instantiate a `DummyValidator` in line 4 and use it to validate a simple string in line 5. It's as simple as that. But we can also take a closer look at the `ValidationResult` that the `DummayValidator` produces

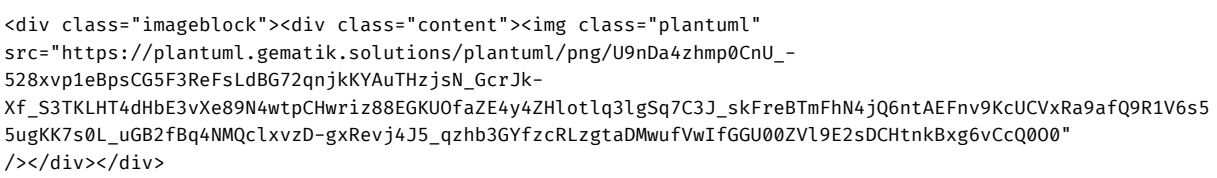
```
@Test
void shouldValidateSimpleString() {
    String content = "Hello Bbriccs!";
    ValidatorFhir validator = new DummyValidator(FhirContext.forR4());

    ValidationResult vr = validator.validate(content);
    assertTrue(vr.isSuccessful());
    assertEquals(1, vr.getMessages().size());
    assertEquals(ResultSeverityEnum.INFORMATION, vr.getMessages().get(0).getSeverity());

    vr.getMessages().forEach(m -> System.out.println(m.getSeverity() + ": " + m.getMessage()));
}
```

And we will see, that the `DummyValidator` produces the following `INFORMATION` message. This is a good way to ensure that the validation is working as expected.

```
# output of the shouldValidateSimpleString unit test
INFORMATION: Information provided by DummyValidator
```

Perimeter-Bricks

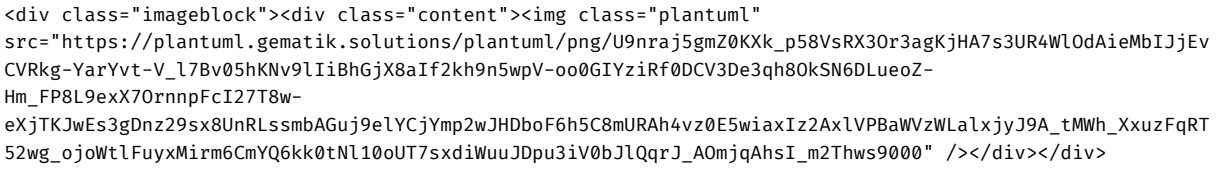
A *perimeter* is a boundary that surrounds a particular area. In the context of TI, we also have a perimeter. It is the boundary that surrounds the TI and separates it from the outside world. The perimeter is the first line of defence against unauthorised access to the TI. This perimeter is usually crossed by a combination of a smart card, a card terminal and a Konnektor.



Smartcards Brick

When testing the Gematik Telematikinfrastruktur, we often have to deal with smart cards. These are used in the TI for authorization, authentication and the creation of signatures, etc. With the goal of (test-)automation, we often pursue the approach of using “virtual images” of the smart cards. However, dealing with certificates and especially with the raw cryptography files can be really challenging. To simplify this process, we have created the Smartcards Brick.

This brick provides a simple and easy-to-use API for loading and dealing with smart cards.



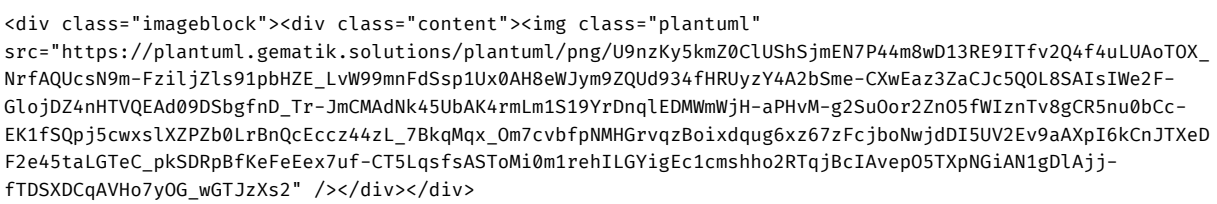
Smartcards API

For example, when dealing with raw .p12 files or `X509Certificates`, it is very easy to mix up the certificates.

Imagine you have an API with the following signatures:

```
public byte[] signData(X509Certificate certificate, byte[] data);
public boolean verifySignature(X509Certificate certificate, byte[] data, byte[] signature);
```

When you encounter this API, you may ask yourself: *"Which certificate do I need to use for which operation?"* Wouldn’t it be nice to have a more structured way of dealing with certificates? To solve this kind of problem, the `smartcard-api-brick` introduces classes for each type of smartcard.



So, instead of dealing with the raw files, you can now treat your "virtual

smartcards" as Java objects in your code. The first example could be refactored to something more intuitive like:

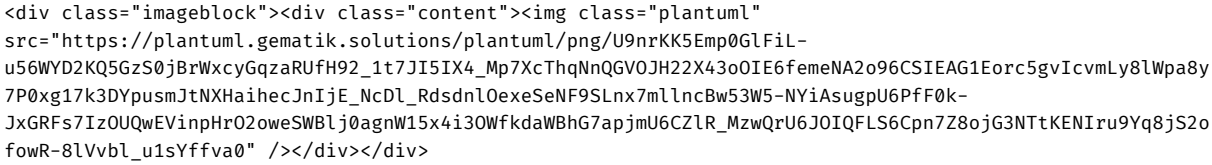
```
public byte[] signData(Hba hba, byte[] data);
public boolean verifySignature(Smartcard smartcard, byte[] data, byte[] signature);
```

From a user perspective, you can provide a clean API for smartcard-related operations without worrying about which certificate to use.

```
Egk egk; // we will get there soon
Hba hba; // we will get there soon
byte[] data; // coming from somewhere else

var signature = signData(hba, data);
var isValid = verifySignature(egk, data, signature);

// the compiler will prevent you from doing this
var s = signData(ega, data);
// but not from this
var v = verifySignature(hba, data, signature);
```

Smartcard-Archiv

Imagine having an archive of all the virtual smartcards you might need, with a simple API to retrieve them easily. That's exactly what the `SmartcardArchive` does.



Before we can use the `SmartcardArchive`, we need to initialise it with the smartcards file we want. You can do this by loading your own smartcards from the file system:

```
var smartcardsImage = new File("your/path/to/smartcards.json");
var sca = SmartcardArchive.from(smartcardsImage);
```

Alternatively, we can save the smartcards file as Java resources (`resources/smartcards/smartcards.json`) and then load them into the `SmartcardArchive` as follows:

```
var sca = SmartcardArchive.fromResources();
```

By convention, `resources/smartcards/smartcards.json` is the default location for the smartcards file.

The smartcards can then be loaded from the `SmartcardArchive` using various criteria:

```
Egk egk0 = sca.getEgk(0);
```

```
Egk egk1 = sca.getEgkByICCSN("80276883110000113311");
Egk egk2 = sca.getEgkByKvnr("X110407071");

Hba hba0 = sca.getHba(0);
Hba hba1 = sca.getHbaByICCSN("80276001011699901501");

SmcB smcb0 = sca.getSmcB(0);
SmcB smcb1 = sca.getSmcBByICCSN("80276001011699900861");
```

The `SmartcardArchive` also offers the option of loading a smartcard based on the specific type and ICCSN:

```
Egk egk1       = sca.getByICCSN(Egk.class, "80276883110000113311");
Smartcard egk2 = sca.getSmartcardByICCSN(SmartcardType.EGK, "80276883110000113311");

Hba hba1       = sca.getByICCSN(Hba.class, "80276001011699901501");
Smartcard hba2 = sca.getSmartcardByICCSN(SmartcardType.HBA, "80276001011699901501");

SmcB smcb0     = sca.getByICCSN(SmcB.class, "80276001011699900861");
Smartcard hba2 = sca.getSmartcardByICCSN(SmartcardType.SMC_B, "80276001011699900861");
```

When called via `getSmartcardByICCSN`, the smartcard is returned as the base class `Smartcard` because the concrete type of the smartcard is not syntactically known here. Whenever possible, the concrete and typed methods should be used in order to maintain type safety.

However, the `SmartcardArchive` can of course only load and issue smartcards that are configured. This means that the following two calls lead to errors at runtime:

```
// exeeding index
assertThrows(IndexOutOfBoundsException.class, () -> sca.getEgk(100));

// unknown ICCSN
assertThrows(SmartcardNotFoundException.class, () -> sca.getEgkByICCSN("123"));
```

We strongly advise against using the loading of smartcards via the index as the standard method.

Both error cases can be handled at runtime as follows:

```
// gives you the amount of known EGK smartcards
int idx = 100;
int amountEgks = sca.getConfigsFor(SmartcardType.EGK).size();
if (amountEgks <= idx) {
    // do something about it
}

// gives you an empty optional because ICCSN "123" is unknown
Optional<Smartcard> opt = sca.getByICCSN("123");
if (opt.isEmpty()) {
    // do something about it
}
```

Konnektor

The "Konnektor" is basically the gateway into the TI. It is a hardware device that is connected to the practice’s network and is used to communicate with the TI. The Konnektor is the central point of the practice’s network and is responsible for the secure transmission of data to the TI.

<img class="plantuml"
src="https://plantuml.gematik.solutions/plantuml/png/U9oLb4zF0Z4GnU_z54qyDo4S3I4WGc8yI8AUZ8VI7TPc-
y_fBCOOlhi5T-
EoK9Kzx67crzVtPYSxYIIHQcjOhBGB4gNbojlW7JXQqOS1Zg18kj80EyVmJyRJSoDLLQAlNJ7pnkDef1q1Ecd6OturdSy6j_FpN33IEIZ
wcCS2y1HPB8R3mO3j6FinSEzJjoAFZPNdgYs8zoXaq6ADMbKll6kgJogZarP3VZUZtr02H0JSQWLv5GJh2Vv2TymbYgCkNudxJTxcTIVH
wc6wvDFbNJVlJ6Bn26XrqcqZgrGJr1GxOasN1Fgrf-EKl3FPLeNY2NCKWKcrj_MKWhBzYmi36nBZx2XJcZf2RE13-
3sTyzlYBAidwd1yNNzu26-
rngIHb_YF3IkTB0zECdjmOgIVEw3VwW8YRmwo_A9ShiNEVQn6vfTKsWO3DXr9uybinKKYZ4t05UdtyGMe_du5" />

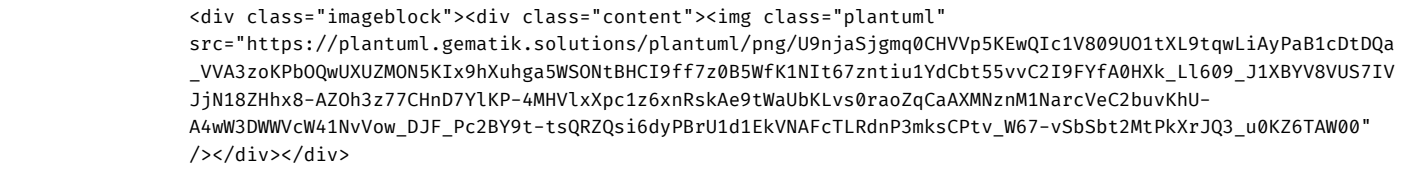
Configuration-Bricks

Configuration is crucial for most types of software, including test suites. It is a way of adapting the behaviour of the software to different environments, or changing the behaviour of the software without changing the code. Configuration can be done in many ways, for example using environment variables, system properties or configuration files. The `configuration-bricks` are designed to help you deal with common problems.

Feature-Toggle

A feature toggle module allows developers to enable or disable certain functions or code sections in an application. This is particularly useful for the gradual introduction of new features. Especially in test suites, we often have the case that certain features go through different stages and need to be activated or deactivated in different environments.

The `feature-toggle-brick` offers a flexible solution by allowing functions to be enabled or disabled via system properties and environment variables.



To increase maintainability in the code, the `feature-toggle-brick` offers an abstraction with a simple API to read feature toggles from the system without directly accessing system properties or environment variables. This reduces the dependency on the respective "key" (whether it is a system property or an environment variable) to a minimum. It not only focuses on the abstraction of system properties/environment variables but also offers a concept to configure feature toggles simultaneously via system properties and environment variables.

Use it when:

- You have many feature toggles in your application
- You have to use specific feature toggles in many different places of your code
- You need to use SystemProperties and EnvironmentVariables simultaneously to control your feature toggles

Don’t use it when:

- You have only a few feature toggles in your application and the `feature-toggle-brick` is not already in your classpath

Using basic Feature-Toggles

A decisive advantage of the `feature-toggle-brick` is that feature toggles can be set both via system properties and via environment variables. The configuration via system properties is preferred, if these are not set, the environment variables are used and in the very last case (if neither system properties nor environment variables are set) the default value is used.

Let’s assume we have the application `MyApplication` which requires a Boolean toggle:

```
class MyApplication {
  public static void main(String[] args){
    val featureConf = new FeatureConfiguration();
    boolean isFeatureActive = featureConf.getBooleanToggle("myFeature.isActive");
    // Alternative mit meinem eigenen Default-Wert: false ist der Default-Wert für den Default-Wert
    // boolean isFeatureActive = featureConf.getBooleanToggle("hello.boolean", false);

    if(isFeatureActive){
      System.out.println("Feature is active");
    } else {
      System.out.println("Feature is NOT active");
    }
  }
}
```

Now we can set the feature toggle via system properties or environment variables and start the application:

```
# verwendet den Default-Wert weil kein Feature-Toggle gesetzt ist
java MyApplication
>> Feature is NOT active

# setze das Feature-Toggle über System-Properties
java -DmyFeature.isActive=true MyApplication
>> Feature is active

# setze das Feature-Toggle über Umgebungsvariablen
export MYFEATURE_ISACTIVE=YES
java MyApplication
>> Feature is active

# Umgebungsvariable weiterhin gesetzt
# das Feature-Toggle wird über System-Properties überschrieben
echo $MYFEATURE_ISACTIVE
java -DmyFeature.isActive=false MyApplication
>> Feature is NOT active
```

This small demonstrator already shows the basics of the `feature-toggle-brick`:

- Feature toggles can be set via system properties and environment variables
- Configuration via system properties has priority over configuration via environment variables
- The default value is used if neither system properties nor environment variables are set. This avoids unexpected errors and we can always start the application in a defined state without additional configurations.

Then there is the nomenclature of the toggles themselves. The `feature-toggle-brick` ensures that the feature toggles are always derived from a uniform formation rule and always follow a fixed scheme.

The following derivation rules apply to the system properties and environment variables:

- the feature toggle `myFeature.isActive` is set as a system property with `-DmyFeature.isActive=true` and therefore has exactly the same name
- the feature toggle `myFeature.isActive` is set as an environment variable with `MYFEATURE_ISACTIVE=YES` and is formed via `key.toUpperCase().replace(“.”, “_”)`.

- A feature toggle (if queried via a string) is always queried via the name (e.g. `myFeature.isActive`) via the API of the `FeatureConfiguration`.

Implementing custom Feature-Toggles

In the last tutorial we saw how a feature toggle can be queried via its name. The name of the feature toggle is passed as a string and the `FeatureConfiguration` takes care of the resolution and the specific mapping to the respective data types of the feature toggle. Other common data types such as `Integer`, `Double`, `String` or generally for enumerations are already implemented and can be used directly.

Although this allows us to ensure a consistent nomenclature of feature toggles, we still have the problem that feature toggles are usually extremely difficult to maintain (if they are queried by string). This type of query is therefore only recommended if a feature toggle is only queried at a single point. For more complex feature toggles, or those that are to be used in many places, the `feature-toggle-brick` also offers the option of creating user-defined feature toggles. This allows the actual name of a toggle to be encapsulated in a single place, which significantly improves the maintainability and readability of the code.

Task: Suppose we have a feature toggle with multiple states called `myFeature.state` which is also needed in many places.

This is the ideal use case for a user-defined feature toggle.

Step 1: implement an enumaration `MyFeatureState` to represent the states of the feature toggle.

```
class MyApplication {
  public static void main(String[] args){
    val featureConf = new FeatureConfiguration();
    MyFeatureStateToggle isFeatureActive = featureConf.getEnumToggle("myFeature.state",
MyFeatureState.class, MyFeatureState.STATE_ONE);
    System.out.println("Feature state is: " + isFeatureActive.getState());
  }

  enum MyFeatureState {
    STATE_ONE("first"),
    STATE_TWO("second"),
    STATE_THREE("third");

    private final String state;

    MyFeatureState(String state){
      this.state = state;
    }

    public String getState(){
      return state;
    }
  }
}
```

Then we can use the feature toggle directly in the simplest case:

```
java -DmyFeature.state=STATE_THREE MyApplication
```

```
>> Feature state is: third
```

However, there is still the problem of maintainability with "string addressing". This can be solved by using a user-defined feature toggle and the feature toggle can also be provided with additional functionality.

Step 2: implement the class `MyFeatureStateToggle` which implements the `FeatureToggle` interface for the feature toggle `MyFeatureState`.

```
class MyApplication {
  public static void main(String[] args){
    val featureConf = new FeatureConfiguration();
    MyFeatureState isFeatureActive = featureConf.getToggle(new MyFeatureStateToggle());
    System.out.println("Feature state is: " + isFeatureActive.getState());
  }

  public static class MyFeatureStateToggle implements FeatureToggle<MyFeatureState> {

    @Override
    public String getKey() {
      return "myFeature.state";
    }

    @Override
    public Function<String, MyFeatureState> getConverter() {
      return this::mapState;
    }

    public MyFeatureState mapState(String value) {
      if (StringUtils.isNumeric(value)) {
        val index = Integer.parseInt(value) % MyFeatureState.values().length;
        return MyFeatureState.values()[index];
      } else {
        return Arrays.stream(MyFeatureState.values())
            .filter(e -> e.getState().equalsIgnoreCase(value) || e.name().equalsIgnoreCase(value))
            .findFirst()
            .orElse(getDefaultValue());
      }
    }

    @Override
    public MyFeatureState getDefaultValue() {
      return MyFeatureState.STATE_ONE;
    }
  }

  enum MyFeatureState {
    STATE_ONE("first"),
    STATE_TWO("second"),
    STATE_THREE("third");

    private final String state;

    MyFeatureState(String state){
      this.state = state;
    }

    public String getState(){
      return state;
    }
  }
}
```

With this implementation, the entire feature toggle is encapsulated in a single

class and can be queried at any point.

```
# mapping über den Enumerationsnamen
java -DmyFeature.state=STATE_THREE MyApplication
>> Feature state is: third

# mapping über den State-Wert der Enumeration
java -DmyFeature.state=second MyApplication
>> Feature state is: second

# mapping über den Enumerationsindex
java -DmyFeature.state=2 MyApplication
>> Feature state is: third

# mapping über einen rollenden Enumerationsindex (Überlauf mit modulo)
java -DmyFeature.state=42 MyApplication
>> Feature state is: first
```

The nomenclature of the feature toggles is retained and the maintainability of the code is significantly increased. Furthermore, the feature toggle can also be provided with additional functionality, such as the conversion of numerical values into enumerations or the use of descriptive values (instead of enumeration names):

This concept can be used to build both simple and extremely complex and, above all, reusable feature toggles.

Utility-Bricks

TODO: descreibe the `utility-bricks` *here*

CLI-Bricks

TODO: descreibe the `cli-tooling-bricks` *here*

Appendix A: Release Notes

Release 0.5.0

RESTful-Bricks

- Extend `restful-bricks` for ePA-Bricks

Release 0.4.0

FHIR-Bricks

- Preference for newer SID system identifiers as defaults and abandonment of old system identifiers

Release 0.2.0

FHIR-Bricks

- Extend `fhir-bricks` for integration with `erp-e2e-testsuite`

Utility-Bricks

- Implement the `vsdm-check-digit-brick` for validating VSDM++ check digits

Release 0.1.10

FHIR-Bricks

- Extend `fhir-de-basisprofil-r4-brick` with `ASK` and `ATC` codings

RESTful-Bricks

- Implement the `raw-http-brick` for encoding and decoding raw HTTP messages

Configuration-Bricks

- Implement the `feature-toggle-brick` for reusable feature toggles

Release 0.1.8

- initial implementation of the core framework