

Bbriccs

gematik GmbH

Version 0.1.10, 2024-11-11

Inhaltsverzeichnis

| | |
|-------------------------------------|----|
| 1. Einleitung | 1 |
| 1.1. Acronym | 1 |
| 1.2. Motivation | 1 |
| 2. Bricks-Module | 3 |
| 2.1. Configuration-Bricks | 3 |
| 2.1.1. Feature-Toggle | 3 |
| 2.1.2. FHIR-Bricks | 8 |
| 2.1.3. Perimeter-Bricks | 9 |
| 2.1.4. RESTful-Bricks | 13 |
| 2.1.5. CLI-Bricks | 14 |
| 2.1.6. Utility-Bricks | 15 |
| Appendix A: Release Notes | 16 |

1. Einleitung

B²ric²s (**briks**) offers reusable building bricks for flexible and maintainable test suites based on clean code principles.

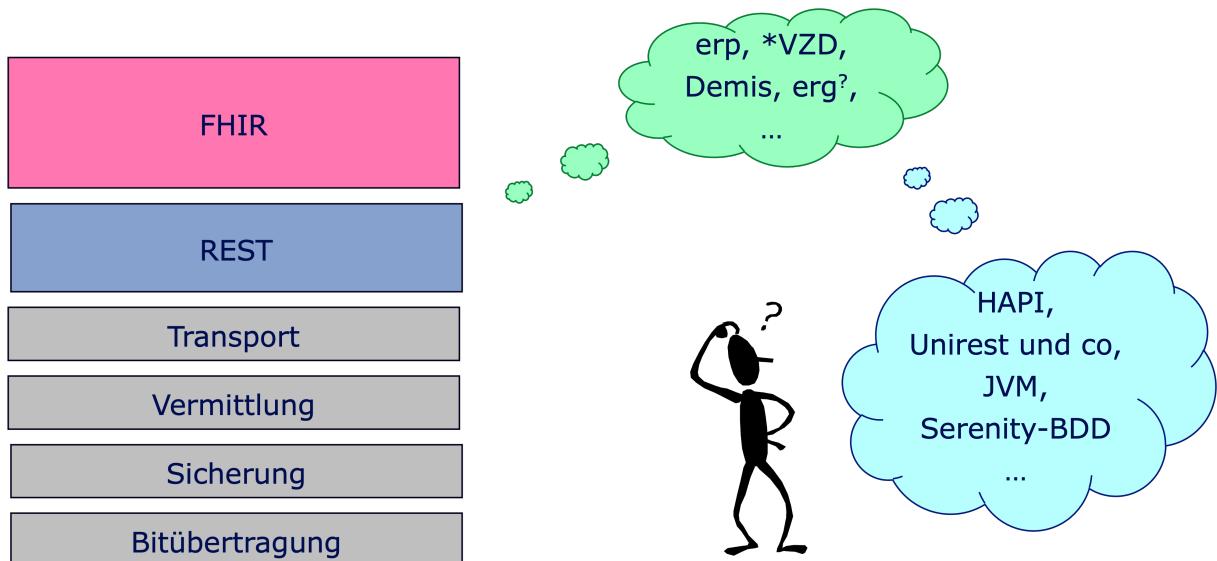
1.1. Acronym

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

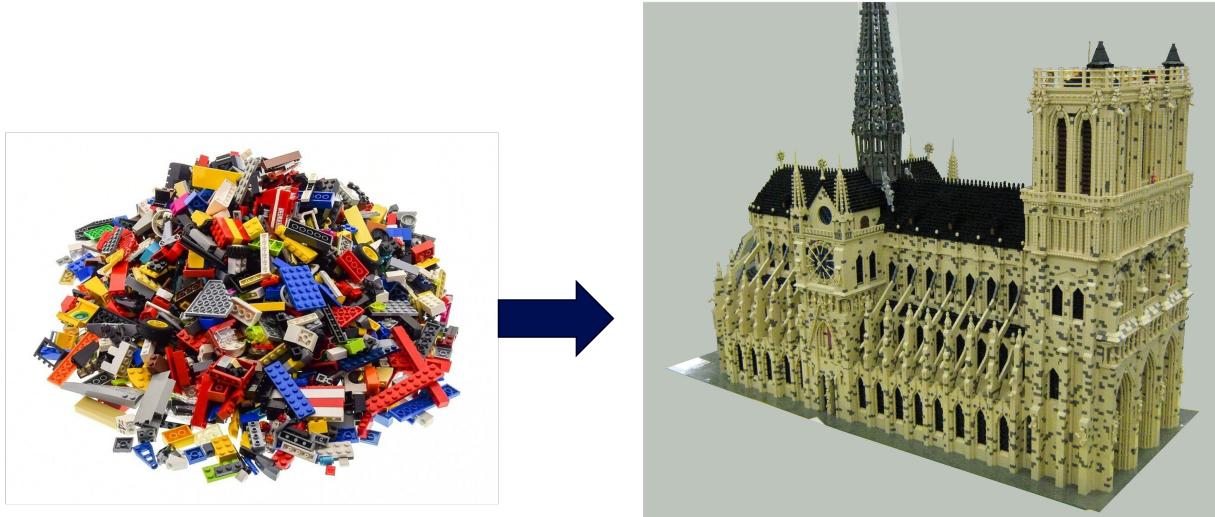
- Pronunciation: simply **briks** based on the idea clamping blocks
- Spelling: in the simplest case just **bricks** or **bbriccs** as a mnemonic for the word mark
- Code: the spelling **bbriccs** is only to be used for package names and namespaces, while **brick** (or **bricks** for plural) shall be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

1.2. Motivation

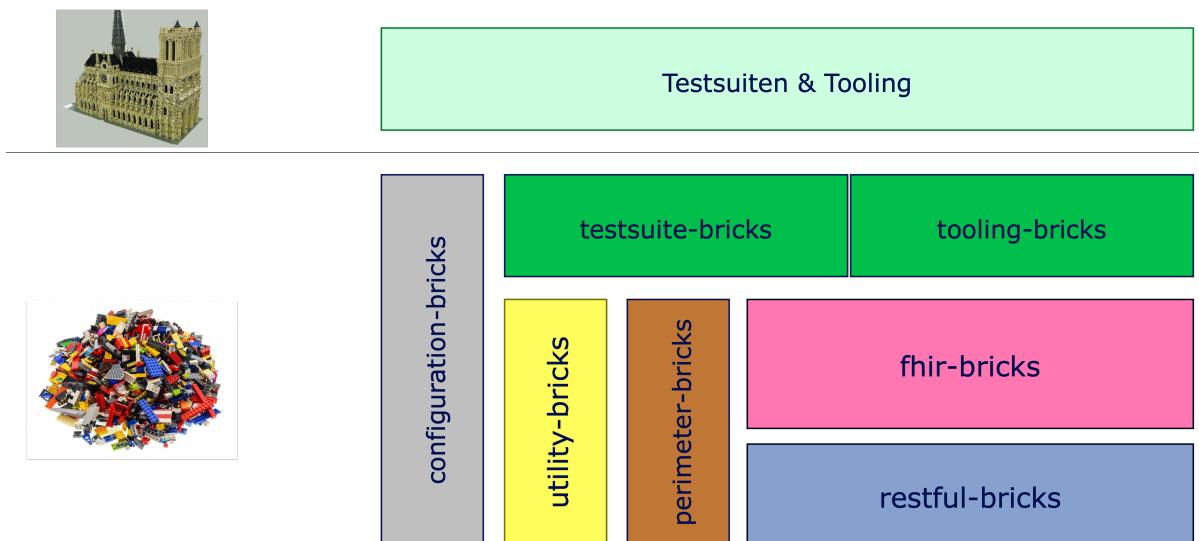
Within gematik GmbH, many of the products developed have a very similar technology stack.



B²ric²s provides a modular construction kit for the implementation of test suites and test tools, which simplifies the reuse of test code and the creation of test suites.



Thanks to the modular design, test suites can be assembled from existing modules and adapted to the respective requirements.



2. Bricks-Module

Nach dem Baukästenprinzip ist B²ric²s in mehrere separate Teile gruppiert...

2.1. Configuration-Bricks

TODO: Beschreibung what are Configuration-Bricks

2.1.1. Feature-Toggle

Ein Feature-Toggle-Modul ermöglicht es Entwicklern, bestimmte Funktionen oder Codeabschnitte in einer Anwendung ein- oder auszuschalten. Dies ist besonders nützlich für die schrittweise Einführung neuer Features. Besonders bei Testsuiten haben wir oftmals den Fall, dass bestimmte Features unterschiedliche Stages durchlaufen und in unterschiedlichen Umgebungen aktiviert oder deaktiviert werden müssen.

Das `feature-toggle-brick` bietet eine flexible Lösung, indem es ermöglicht, Funktionen über die System-Properties und Umgebungsvariablen zu aktivieren oder zu deaktivieren.

```
<div class="imageblock"><div class="content"></div></div>
```

Um die Wartbarkeit im Code zu erhöhen, bietet das `feature-toggle-brick` eine Abstraktion mit einer einfachen API an Feature-Toggles aus dem System auszulesen ohne dabei direkt auf System-Properties oder Umgebungsvariablen. Dadurch lässt sich die Abhängigkeit zum jeweiligen "Key" (sei es ein System-Property oder eine Umgebungsvariable) auf ein Minimum reduzieren. Dabei wird nicht nur die Abstraktion von System-Properties/Umgebungsvariablen fokussiert, sondern auch ein Konzept angeboten, um die Feature-Toggles sowohl über System-Properties als auch über Umgebungsvariablen gleichzeitig zu konfigurieren.

Feature-Toggles setzen

Ein entscheidender Vorteil des `feature-toggle-brick` ist es, dass Feature-Toggle sowohl über System-Properties als auch über Umgebungsvariablen gesetzt werden können. Dabei wird die Konfiguration über System-Properties bevorzugt, falls diese nicht gesetzt sind, wird auf die Umgebungsvariablen zurückgegriffen und im aller letzten Fall (wenn weder System-Properties noch Umgebungsvariablen gesetzt sind) wird der Default-Wert verwendet.

Angenommen wir haben die Applikation `MyApplication` die ein Boolean-Toggle benötigt:

```
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");
    }
}

```

Nun können wir das Feature-Toggle über System-Properties oder Umgebungsvariablen setzen und die Applikation starten:

```

# 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

```

Mit diesem kleinen Demonstrator werden bereits die Grundlagen des **feature-toggle-brick** demonstriert:

- Feature-Toggles können über System-Properties und Umgebungsvariablen gesetzt werden
- Die Konfiguration über System-Properties hat Vorrang vor der Konfiguration über Umgebungsvariablen
- Der Default-Wert wird verwendet, wenn weder System-Properties noch Umgebungsvariablen gesetzt sind. Damit werden unerwartete Fehler vermieden und wir können die Applikation immer in einem definierten Zustand ohne zusätzliche Konfigurationen starten.

Dann wäre da noch die Nomenklatur der Toggles selbst. Das **feature-toggle-brick** sorgt dafür, dass die Feature-Toggles stets aus einer einheitlichen Bildungsregel abgeleitet werden und immer einem festen Schema folgen.

Dabei gelten die folgenden Ableitungsregeln für die System-Properties und Umgebungsvariablen:

- das Feature-Toggle `myFeature.isActive` wird als System-Property mit `-DmyFeature.isActive=true` gesetzt und hat damit exakt den gleichen Namen
- das Feature-Toggle `myFeature.isActive` wird als Umgebungsvariable mit `MYFEATURE_ISACTIVE=YES` und wird dabei über `key.toUpperCase().replace(".", "_")` gebildet.
- über die API der `FeatureConfiguration` wird ein Feature-Toggle (wenn über

einen String abgefragt) immer über den Namen (z.B. `myFeature.isActive`) abgefragt.

Benutzerdefinierte Feature-Toggles

Im letzten Beispiel haben wir gesehen, wie man ein Feature-Toggle über seinen Namen abfragen kann. Dabei wird der Name des Feature-Toggles als String übergeben und die `FeatureConfiguration` kümmert sich um die Auflösung und das konkrete Mapping zum jeweiligen Datentypen des Feature-Toggles. So sind z.B. auch weitere gängige Datentypen wie `Integer`, `Double`, `String` oder auch allgemein für Enumerationen bereits implementiert.

Damit können wir zwar schon die konsistente Nomenklatur der Feature-Toggles gewährleisten, haben aber weiterhin das Problem, dass Feature-Toggles tendenziell(wenn über String abgefragt), extrem schlecht wartbar sind. Diese Art der Abfrage ist daher nur empfehlenswert, wenn ein Feature-Toggle nur an einer einzigen Stelle abgefragt wird. Für komplexere Feature-Toggles, oder solche die an vielen Stellen verwendet werden müssen, bietet das `feature-toggle-brick` auch die Möglichkeit, benutzerdefinierte Feature-Toggles zu erstellen. Damit kann der tatsächliche Name eines Toggles in einer einzigen Stelle gekapselt werden, wodurch die Wartbarkeit und Lesbarkeit des Codes deutlich erhöht wird.

Angenommen wir haben ein Feature-Toggle mit mehreren Zuständen mit dem Namen `myFeature.state` welches zusätzlich an vielen Stellen benötigt wird. Der Zustand dafür kann über eine Enumeration `MyFeatureState` abgebildet werden und folgendermaßen abgefragt werden:

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

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

        private final String state;

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

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

Dann können wir das Feature-Toggle bereits so auch im einfachsten Fall direkt nutzen:

```
java -DmyFeature.state=STATE_THREE MyApplication
>> Feature state is: third
```

Allerdings besteht hier weiterhin das Problem der Wartbarkeit mit der "String-Adressierung". Diese lässt sich durch die Verwendung eines benutzerdefinierten Feature-Toggles lösen und darüber hinaus lässt sich das Feature-Toggle auch mit zusätzlicher Funktionalität versehen:

```
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;
    }
}
```

Mit dieser Implementierung ist das gesamte FeatureToggle in einer einzigen Klasse gekapselt und kann an beliebiger Stelle abgefragt werden. Dabei wird die Nomenklatur der Feature-Toggles beibehalten und die Wartbarkeit des Codes deutlich erhöht. Darüber hinaus kann das Feature-Toggle auch mit zusätzlicher Funktionalität versehen werden, wie z.B. die Umwandlung von numerischen Werten in Enumerationen oder auch die Verwendung von sprechenden Werten

(anstatt der Enumerationsnamen):

```
# 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
```

Mit diesem Konzept lassen sich sowohl einfache als auch extrem komplexe und vor allem wiederverwendbare Feature-Toggles bauen. <<< :leveloffset: -1

2.1.2. FHIR-Bricks

Mit [HL7 FHIR](#)

TODO: Beschreibung what is FHIR-Bricks

```
<div class="imageblock"><div class="content"></div></div>
```

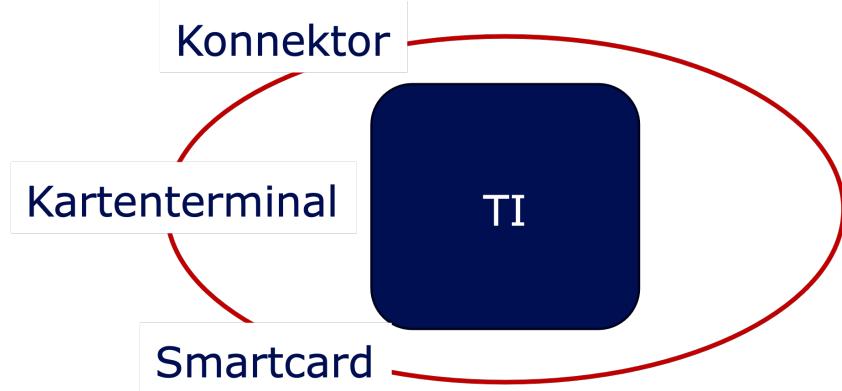
FHIR-Validator

Was ist der FHIR-Validator und wie funktionieren die Implementierungen...
TODO

```
<div class="imageblock"><div class="content"></div></div>
```

2.1.3. Perimeter-Bricks

Was sind die TI-Perimeter-Bricks... TODO



Smartcards-Brick

Smartcards mittels `smartcard-brick`.... TODO

```
<div class="imageblock"><div class="content"></div></div>
```

Smartcards

Die Smartcards sind folgendermaßen aufgebaut:

```
<div class="imageblock"><div class="content"></div></div>
```

Smartcard-Archiv

Das `SmartcardArchive` ist ein Archiv, in dem die Smartcards verwaltet werden.

```
<div class="imageblock"><div class="content"></div></div>
```

Bevor man das `SmartcardArchive` nutzen kann, muss dieses zunächst mit den gewünschten Smartcards initialisiert werden. Hierfür hat man die Möglichkeit eigene Smartcards aus dem Dateisystem zu laden:

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

Alternativ ist es auch möglich, die Smartcard Images als Java-Resources (`resources/smartcards/smartcards.json`) zu speichern und diese dann folgendermaßen in das `SmartcardArchive` zu laden:

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

Anschließend können die Smartcards aus dem `SmartcardArchive` anhand unterschiedlicher Kriterien geladen werden:

```
var egk0 = sca.getEgk(0);
var egk1 = sca.getEgkByICCSN("80276883110000113311");
var egk2 = sca.getEgkByVnNr("X110407071");

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

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

Das `SmartcardArchive` bietet darüber hinaus auch die Möglichkeit eine Smartcard anhand des konkreten Typs und der ICCSN zu laden:

```
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");
```



Bei dem Aufruf über `getSmartcardByICCSN` wird die Smartcard als Basis-Klasse `Smartcard` zurückgegeben, weil hier der konkrete Typ der Smartcard syntaktisch nicht bekannt ist. Es sollen, wenn immer möglich, die konkreten und typisierten Methoden verwendet werden um die Typsicherheit zu bewahren.

Das `SmartcardArchive` kann natürlich aber nur Smartcards laden und herausgeben, die konfiguriert sind. Das bedeutet, dass die folgenden beiden Aufrufe zur Laufzeit zu Fehlern führen:

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

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



Grundsätzlich wird dringend davon abgeraten, das Laden der Smartcards über den Index als Standardmethode zu verwenden.

Beide Fehlerfälle können zur Laufzeit in etwa folgendermaßen behandelt werden:

```
// 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
}
```

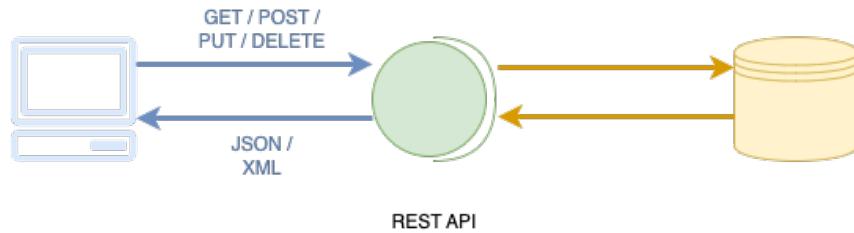
Konnektor

Der Konnektor-Client ist über mehrere Module aufgebaut... TODO

```
<div class="imageblock"><div class="content"></div></div>
```

2.1.4. RESTful-Bricks

Was sind die RESTful-Bricks?



```
<div class="imageblock"><div class="content"></div></div>
```

2.1.5. CLI-Bricks

TODO: Beschreibung what are CLI-Bricks

2.1.6. Utility-Bricks

TODO: Beschreibung what are Utility-Bricks

Appendix A: Release Notes

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