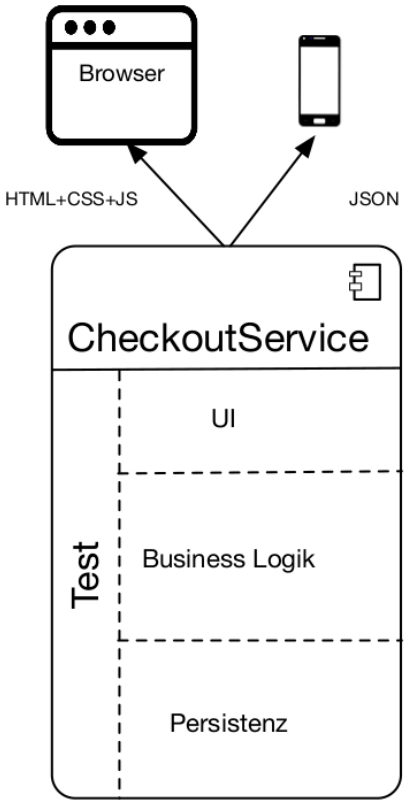
Self-Contained Systems mit Spring Boot



Exemplarische Struktur eines SCS

„Self-Contained …“ was?

Als Self-Contained System (SCS) wird weithin ein Architekturmuster bezeichnet, welches sich durch folgende charakteristischen Eigenschaften auszeichnet: Ein SCS …

- … hat die Hoheit über alle Daten und Prozesse, fachliche wie technische, die einer klar abgrenzbaren Domäne zuzuordnen sind.
- … wird weiterentwickelt, getestet und betrieben von genau einem Team.
- … kommuniziert mit anderen SCS vorzugsweise asynchron.
- … kann Service-Schnittstellen für z. B. mobile Endgeräte anbieten.

- … bietet eine Benutzeroberfläche (UI), mit darunterliegender Geschäftslogik und einer Persistenzschicht für die Daten seiner Domäne an.
- … basiert so wenig wie möglich auf geteilter Infrastruktur.

Und wozu das Ganze?

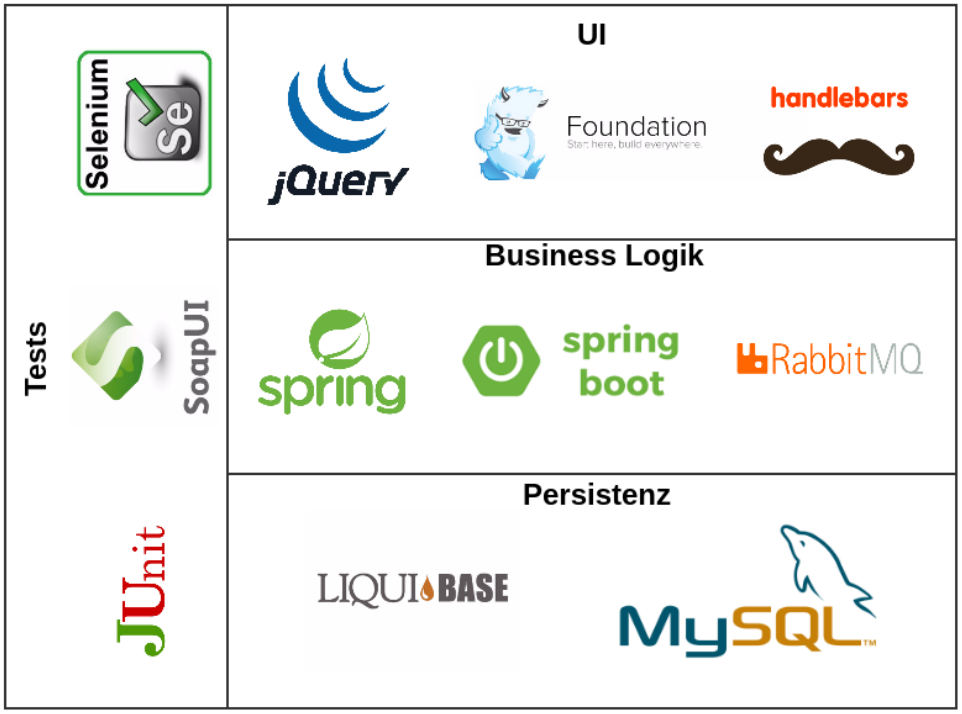
Da der weitere Fokus des Artikels auf der beispielhaften Implementierung des beschrieben Musters liegen soll, möchte ich an dieser Stelle gerne auf eine differenzierte Auseinandersetzung mit den Hintergründen verzichten. Wer sich vorab einen kurzen Überblick über das „Warum?“ verschaffen möchte, dem empfehle ich Why SCS? zu überfliegen. Für ausführlichere Informationen zu den Hintergründen und der Abgrenzung zu monolithischen oder Microservice-Ansätzen lohnt sich ein Blick in die Quellen.

Alternativ ist auch folgender Podcast von InnoQ zum Thema *„Eigenschaften und Umsetzung von SCS“* sehr aufschlussreich (Quelle: https://www.innoq.com/de/podcast/030-scs-teil-1/):

https://www.innoq.com/de/podcast/archive/030-scs-teil-1.mp3

Wie soll das mit Spring Boot funktionieren?

Entsprechend der oben skizzierten Struktur benötigen wir technische Komponenten für die klassischen Schichten UI, Business Logik & Persistenz. Des Weiteren soll der Technologie-Stack Möglichkeiten bereitstellen, jede dieser Komponenten automatisiert testen zu können. Der im Produktteam *„Kunde im Mittelpunkt“* (KIM) eingesetzte Technologie-Stack zur Abbildung dieser Anforderungen setzt sich aus den folgenden Komponenten zusammen, mit Spring Boot als Fundament:



Technologie-Stack im Team KIM

Für das initiale Setup des Spring Boot-Projektes werden im Team KIM beinahe dieselben Dependencies verwendet, wie für eine gewöhnliche Spring MVC-Applikation. Ein gutes Basisprojekt mit den wichtigsten Dependencies hierfür kann in kürzester Zeit über https://start.spring.io/ generiert werden. Alle nötigen Dependencies für ein SCS können der nebenstehenden Abbildung entnommen werden.

Minimal-Set an Dependencies für ein SCS

An dieser Stelle gibt es leider keine Möglichkeit für die Templating-Engine Handlebars alle nötigen Dependencies automatisch mit zu erzeugen, sodass diese manuell ergänzt werden müssen. Des Weiteren verzichte ich hier darauf, das Thema Testing näher zu beleuchten, da hiermit mehrere dedizierte Blog-Artikel gefüllt werden könnten. Schauen wir uns stattdessen einmal genauer an, was es mit den einzelnen Komponenten auf sich hat.

```
{{#if model.generalMessage.message}}
    <div class="{{model.generalMessage.type.value}} callout">
        <p>{{{model.generalMessage.message}}}</p>
        {{#if closable}}
            <button class="close-button" type="button">
                <span>×</span>
            </button>
        {{/if}}
    </div>
{{/if}}
```

Fehler- bzw. Erfolgsmeldung mit Handlebars

Auf oberster Ebene versetzt uns Handlebars dazu in die Lage, dem Benutzer angezeigte Inhalte dynamisch generieren zu lassen. Die Java-Adaption von Handlebars bietet eine einfach Möglichkeit, binnen kürzester Zeit in einer Spring Boot-Applikation derartige Inhalte auszuspielen.

Sind die dynamischen Inhalte, wie z. B. Auftragsdaten, Rechnungen etc. generiert, müssen diese noch in responsiver, benutzerfreundlicher Weise dargestellt werden. Da die Anforderungen an eine endgerätefreundliche Website immer weiter steigen, setzen wir hier auf die Expertise des Foundation-Frameworks, welches bereits umfangreiche Features im Standard enthält, die eine zeitgemäße Visualisierung von Content erlauben. Auf Grund des hohen Bekanntheitsgrades von WebUI-Frameworks a la Twitter-Bootstraps etc. möchte ich hier nicht näher auf Foundation eingehen. Welche andere wichtige Rolle das Frontend spielt, sehen wir noch im Abschnitt zur Integration von SCS in bestehende Systeme.

```
@Controller
@RequiredArgsConstructor(onConstructor = @__(@Autowired))
@RequestMapping("/site/{mandantId}/login/login.html")
public class LoginController {

    @RequestMapping(
            method = GET,
            produces = TEXT_HTML_VALUE)
    public ModelAndView get(@PathVariable final int mandantId) {
        //@formatter:on
        return new ModelAndView( viewName: "login",
                ImmutableMap.of( k1: "model", createModel()));
    }
```

Controller-Definition mit Spring & Lombok

Eine Schicht unterhalb der UI erreichen wir das Backend über Spring-Controller, die HTML bereitstellen, bzw. -RestController für JSON-Inhalte, wenn es sich bei dem aufrufenden Endgerät um einen mobilen Client handelt, der eine native UI

anbietet. Die Controller werden mit Spring-Bordmitteln, wie z. B. Spring Web und Spring Security, abgebildet und stellen die Brücke zur eigentlichen Geschäftslogik dar. Gerät die Geschäftslogik jedoch an ihre Domänengrenzen, wird die Kommunikation mit einem anderen SCS erforderlich.

```
@RabbitListener(bindings = @QueueBinding(
    value = @Queue(
            value = QUEUE,
            durable = "true",
            arguments = @Argument(
                name = "x-dead-letter-exchange",
                value = EXCHANGE_DEADLETTER
            )
    ),
    exchange = @Exchange(
            value = EXCHANGE,
            type = ExchangeTypes.FANOUT,
            ignoreDeclarationExceptions = "true",
            durable = "true"
    )
))
void onKundeUpdate(KundeDto kundeDto) {
```

RabbitListener-Definition mit Deadletter-Exchange

Eine wesentliche Maßgabe hierzu, die sich aus dem SCS-Ansatz ergibt, ist die asynchrone Kommunikation mit anderen SCS, welche wir im Team KIM beispielsweise mit Spring AMQP realisieren. Da Spring Boot das Dogma „*convention over configuration*“ verfolgt, können wir hier von der RabbitAutoConfiguration des Frameworks profitieren, welche bereits eine Menge häufig benötigter Beans mitbringt, wie ein RabbitTemplate, eine ConnectionFactory, einen AmqpAdmin usw. Benutzername, Passwort und Host für die RabbitMQ-Verbindung müssen so nur noch konfigurativ über die application.yaml gepflegt werden. Auch das Lauschen auf Nachrichten, die von RabbitMQ eingehen, kann komplett deklarativ realisiert werden, sodass sich der Umfang an zu schreibenden Codezeilen auf ein Minimum reduzieren lässt.

Alternativ zum ereignisgetriebenen Datenaustausch können Daten auch im Hintergrund synchronisiert und redundant persistiert werden. Hierfür nutzen wir den Scheduling-Mechanismus von Spring innerhalb derselben Applikation, der in einem eigenen Thread per REST-Call an eine andere Applikation Daten repliziert, die das SCS zwar für Aufgaben innerhalb seiner Domäne braucht, worüber es aber nicht die Hoheit (!) hat. Die Applikation ist so auch dann verfügbar (auf einem potenziell veralteten Datenbestand), wenn das SCS, aus dem es seine

Daten bezieht, nicht erreichbar ist.

```
<changeSet id="1" author="dennis.schneider@thalia.de">
    <createTable schemaName="thalia_ecp_login" tableName="benutzer">
        <column autoIncrement="true" name="id" type="BIGINT">
            <constraints nullable="false" primaryKey="true"/>
        </column>
        <column name="kundennummer" type="BIGINT">
            <constraints nullable="false" unique="true"/>
        </column>
        <column name="angelegt_am" type="TIMESTAMP">
            <constraints nullable="false"/>
        </column>
    </createTable>
```

Tabellendefinition mit Liquibase

Um die Daten nun auch noch persistieren zu können, erzeugen und erweitern wir unser Datenbankschema mit Liquibase. Ein entscheidender Vorteil von Liquibase ist, dass es die Anpassungen am Datenbankschema direkt beim Hochfahren der Applikation auf die Datenbank anwendet. Änderungen am Modell können so direkt mit dem Deployment der Applikation automatisch ausgeführt werden, ohne dass ein manueller Eingriff erforderlich ist.

Sind Tabellen und Spalten angelegt, können wir mittels Spring Data JPA auf diese zugreifen. Spring Data übernimmt für uns das gesamte OR-Mapping, nachdem wir die entsprechenden @Entity-, @Table-Annotationen und Co. auf unseren Objektrepräsentationen des Datenbankschemas ergänzt haben. Zuletzt ist für jede Entity noch ein Repository-Interface zu definieren, welches ein Interface vom Typ Spring Data CrudRepository erweitert. Das Framework sorgt anschließend zur Laufzeit dafür, dass ein SimpleJpaRepository erzeugt wird, mit dem auf die entsprechenden Daten zugegriffen werden kann.

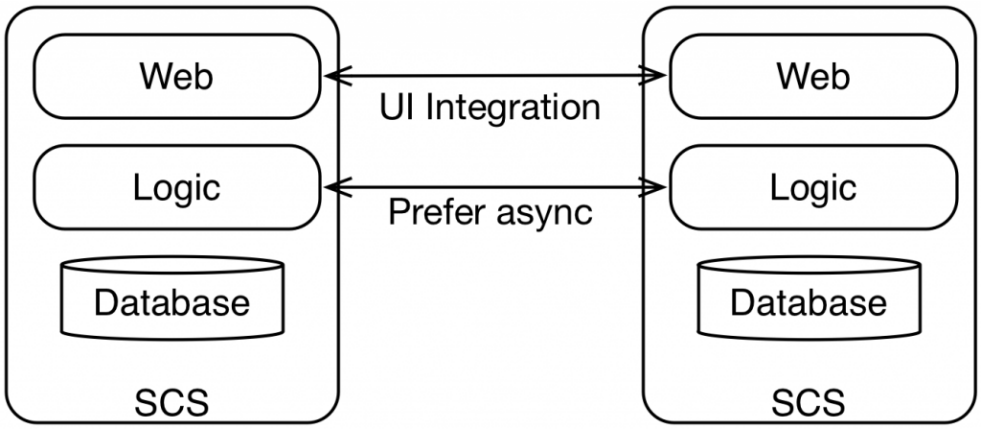
Wie integriert sich ein SCS in bestehende Systeme?

Aus dem vorherigen Abschnitt deutet sich an, dass wir hier über zwei verschiedene Arten von Integration nachdenken müssen:

- Datenintegration
- Prozessintegration

Wie schon beschrieben, erfolgt die Datenintegration vor allem über asynchrone Kommunikation, wie z. B.

- ereignisgesteuerte Mechanismen (RabbitMQ, ActiveMQ, Kafka, …), oder
- regelmäßige bzw. bedarfsgetriebene Abfrage und Persistierung von Daten im Hintergrund (Spring-Scheduler, Quartz, …)



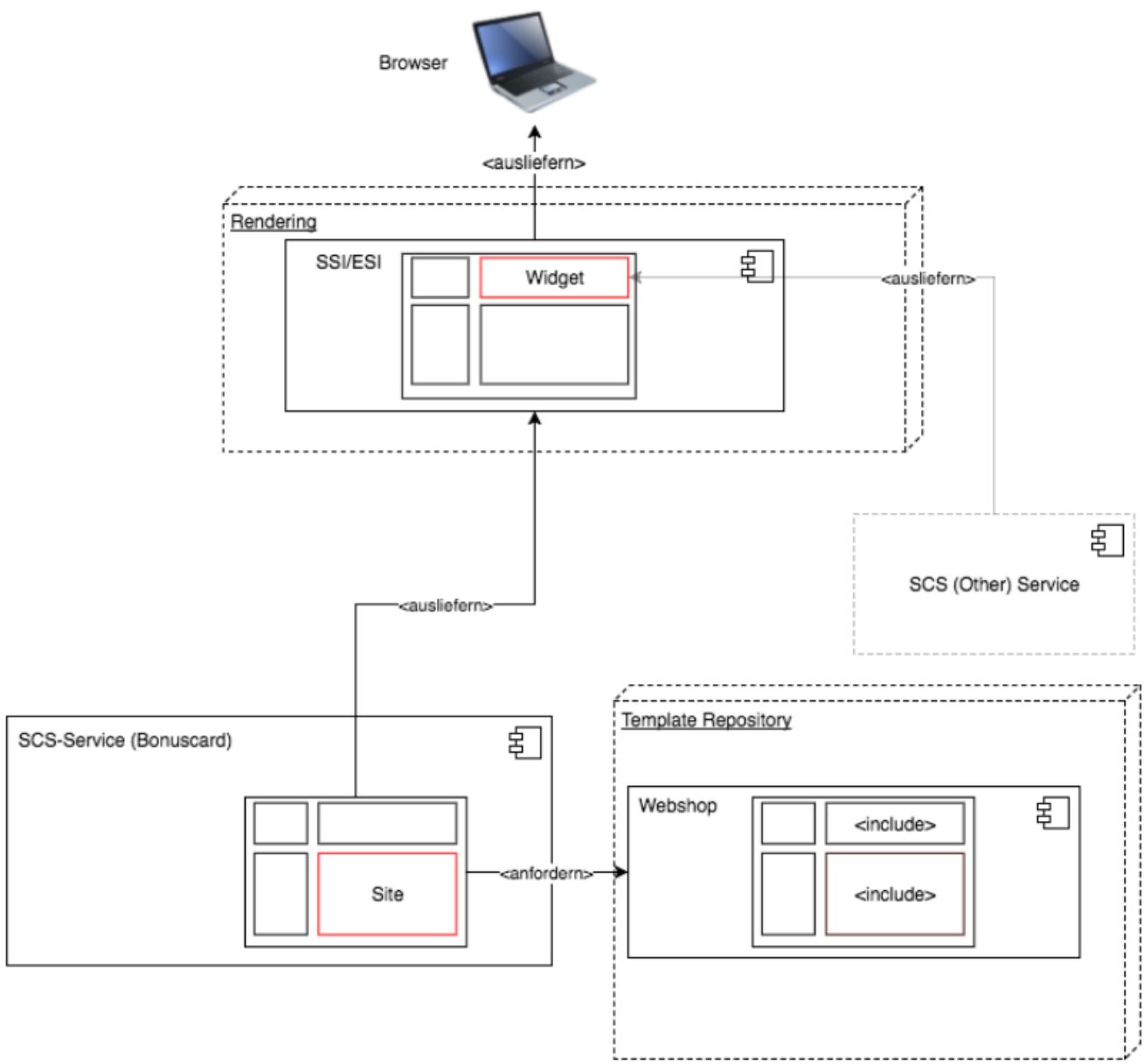
Prozessintegration per UI (Quelle: InnoQ)

Bezogen auf Prozessintegration rückt die UI wieder in den Mittelpunkt. Da backendseitig vor allem asynchron und vorzugsweise innerhalb des eigenen SCS kommuniziert werden soll, findet die domänenübergreifende Prozessintegration über das Frontend statt. Aus Implementierungssicht verwenden wir im Team KIM hierfür die beiden folgenden Ansätze:

1. Hyperlinks, als Einstiegspunkt in eine (andere) Domäne
2. Server-side-includes (SSI) & Ajax, zur Orchestrierung von SCS

Unter der Integration über Hyperlinks kann man sich URLs vorstellen, die auf dem SCS der Zieldomäne terminieren. So landet man bei einem Klick auf „Passwort vergessen?“ auf der Anmeldeseite von Thalia.de beispielsweise auf einem unserer SCS, dessen Domäne der Login bzw. die Registrierung ist.

Sollen hingegen auf einer Website die UIs mehrerer SCS angezeigt werden, orchestrieren wir diese mittels SSI oder Ajax. Über beide Verfahren lassen sich Teilbereiche einer Website mit den Inhalten unterschiedlicher SCS befüllen. Die folgende Abbildung veranschaulicht die Integration mehrerer SCS mittels SSI:



Beispielhafte SCS-Integration mit zentralisiertem Template Repository

Was ist sonst noch zu beachten?

Neben der Implementierung des SCS haben sich in diesem Artikel einige Aspekte am Rande angedeutet, die bei der Realisierung derartiger System nicht zu vernachlässigen sind. Fragestellungen, die hierbei ebenfalls betrachtet werden sollten, könnten sein:

- Wie werden SCS gebaut und deployed?
- Wie lassen sich hierbei Unittests / Schnittstellentests / Integrationstests / End-to-end-Tests einbinden?

- Wie wird eine einheitliche User Experience in SCS sichergestellt?
- Welche Bedeutung hat eine SCS-Architektur für das Entwicklungsteam / die Organisation?

Sicherlich hat diese kurze Liste keinen Anspruch auf Vollständigkeit, deutet aber doch an, auf welche Herausforderungen sich eine Organisationseinheit einstellen muss. Vielleicht widme ich mich ja in Folgebeiträgen einer dieser Fragen bzw., wie wir dieser begegnet sind :-).

Quellen

- http://scs-architecture.org/index.html
- https://www.innoq.com/en/articles/2016/11/self-contained-systems-different-microservices/
- https://jaxenter.de/self-contained-systems-45606
- https://www.heise.de/developer/artikel/Self-contained-Systems-ein-Architekturstil-stellt-sich-vor-3038718.html
- https://spring.io/guides