Vaadin 10 – der Sprung von GWT zu WebComponents

Sven Ruppert

Um GWT durch WebComponents zu ersetzen, wurden alle Komponenten vollständig neu als WebComponents gebaut und dann das Java-API gegen diese WebComponents programmiert.

Lang ist es her, GWT wurde am 17. Mai 2006 von Google veröffentlicht und führte damals zu einem großen Umbruch. Mit GWT konnte der Java-Entwickler Web-Anwendungen sowohl auf der Client- als auch auf der Serverseite realisieren. Für den Entwickler fühlte es sich in gewisser Weise so an, als würde er eine Desktop-Anwendung entwickeln.

Sicherlich gab es auch andere Frameworks, die Ähnliches in Teilen angegangen sind, GWT war allerdings schon sehr mächtig am Markt und ist es vielerorts noch immer. Zum Beispiel gab es ein kleines Unternehmen in Finnland, Turku, mit dem Namen „IT Mill", das ebenfalls ein Framework in dieser Art hatte. Man erkannte damals, dass GWT viele Vorzüge hatte, und ersetzte die hausinterne Implementierung durch GWT. Allerdings ist das für den Entwickler sichtbare API in Java gleich geblieben. Mittlerweile hat sich die Firma umbenannt und ist den meisten unter dem Namen „Vaadin" bekannt. Dieses Jahr war es wieder soweit, GWT wurde durch WebComponents ersetzt *(siehe Abbildung 1)*.

Built for the modern mobile-first web

Heute gibt es drei wichtige Zielplattformen für den Aufbau von UIs für Business-Apps: Web, Android und iOS. Zwar bieten native Android- und iOS-Apps heute wohl das ausgereifteste UX-System, doch das Web hat sich aufgrund verschiedener Eigenschaften als Hauptziel für die meisten Geschäftsanwendungen etabliert.

Es gibt zwei Technologien, die diesen Trend sehr verstärken und die Welt der Geschäftsanwendungen nachhaltig verändern werden.

Progressive Web Applications

Mit Progressive Web Applications (PWA, siehe *„https://vaadin.com/progressive-web-applications")* nähert sich die Benutzererfahrung, die der Anwender einer Web-Anwendung erfährt, immer mehr den nativen Apps auf iOS und Android an. Mit PWA verschmelzen die nativen Apps und die Web Apps in ihrem Verhalten und der Benutzer kann immer weniger Unterschiede feststellen. Die Browser Chrome, Safari, Edge und Firefox bieten die dafür notwendigen Funktionen auf dem Desktop schon heute auf jedem Betriebssystem, auf dem sie ausgeführt werden.

WebComponents

Wofür bis heute Frameworks benötigt worden sind, kommen nun WebComponents *(siehe „https://www.webcomponents.org/introduction")* ins Spiel. Sie verändern das gesamte Ökosystem rund um die Web-Entwicklung, wenn man sich ansieht, wie bisher mit Komponenten gearbeitet worden ist. Ziel ist es, zwischen den verschiedenen Frameworks und Web-Plattformen kompatibel zu bleiben. In der Praxis bedeutet das, dass in einem Projekt WebComponents verschiedener Hersteller zusammen verwendet werden können.

Hier schließt sich der Kreis wieder. Alle Vaadin-Komponenten wie zum Beispiel der Button oder das Grid können als WebCompo-

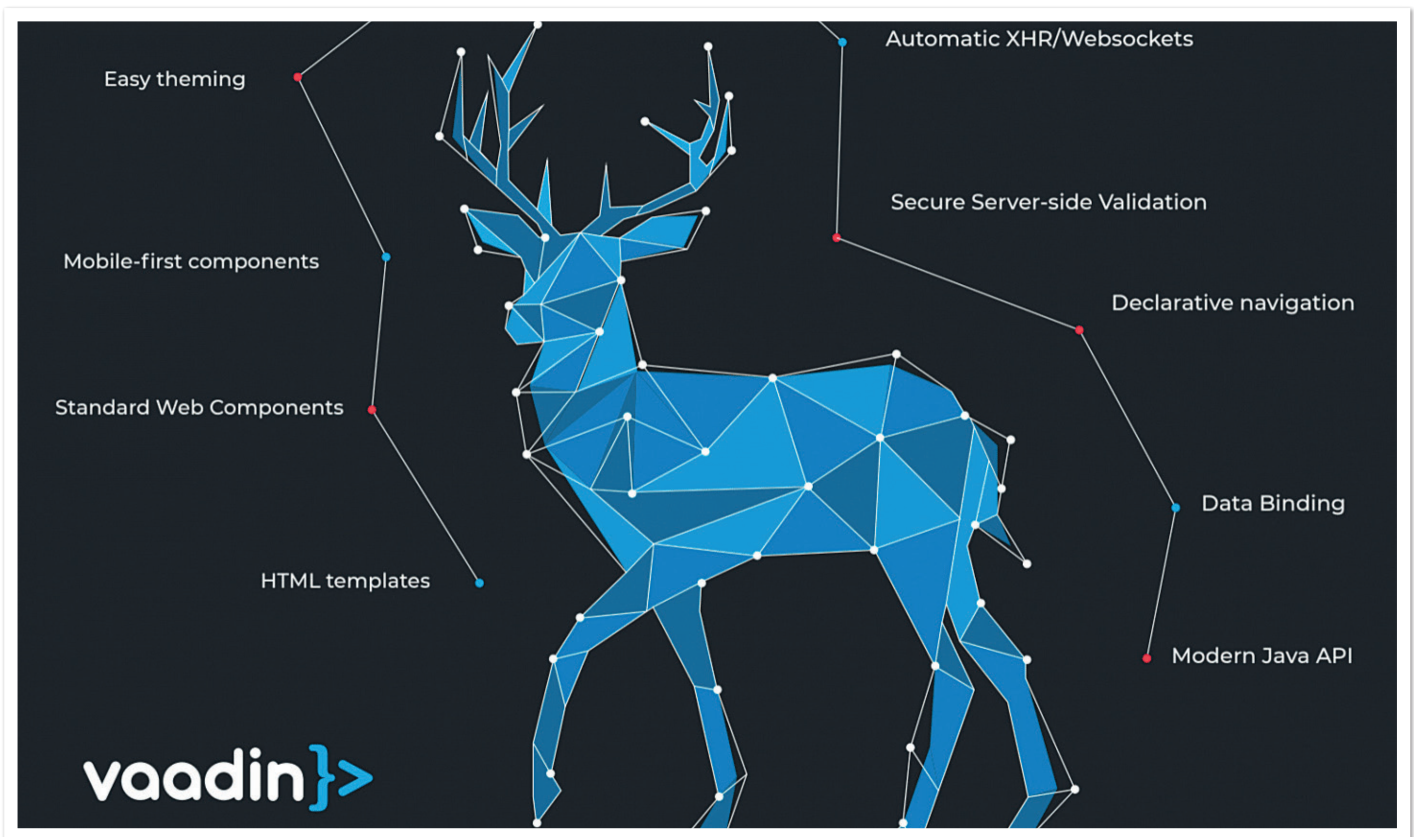

Abbildung 1: Das neue Vaadin

nent beliebig mit anderen WebComponents anderer Hersteller und Frameworks, die mit WebComponents umgehen können, kombiniert werden. Damit ist es nun seit Vaadin 10 möglich, mit Vaadin alleine eine rein clientseitige Anwendung zu entwickeln.

Client- und serverseitige Entwicklung

Bei der Entwicklung von Web-Anwendungen kann man den Fokus auf die Client- und/oder Serverseite legen. Je nachdem, worauf die Web App hin optimiert werden soll. Da es nicht die eine einzige richtige Antwort gibt, wurde Vaadin 10 so konzipiert, dass die Entwicklungsplattform sowohl die clientseitigen als auch die serverseitigen Technologien unterstützt, um Anwendungsentwicklern die Qual der Wahl zu lassen.

```
```xml
<vaadin-date-picker
  label="Select birth date"
  value="[[user.birthDate]]"
  required>
</vaadin-date-picker>
```

```java
DatePicker bd = new DatePicker("Select birth date");
bd.setValue(user.getBirthDate());
bd.setRequired(true);
```
```

Listing 1

Dies sollte nicht bedeuten, dass die Plattform so tun würde, als ob jede Anwendung die gleiche Architektur oder Teile hätte. Stattdessen stellt die Plattform Teile bereit, die gut zusammenarbeiten, allerdings unabhängig voneinander verwendet werden können; die Entwickler wählen die gewünschten Teile selbst aus.

Aufgrund dieser dualen Natur der Plattform können Teams ihre Kompetenzen nutzen und eine nahtlose Zusammenarbeit zwischen Web- und Java-Entwicklern aufbauen. *Listing 1* zeigt ein Beispiel aus beiden Welten.

Vaadin 10 – Flow

Zum Java-Teil: Das, was der Entwickler unter dem Namen „Vaadin-Framework" (Vaadin 8) kennt, heißt nun „Flow". Es ist Teil der Vaadin-Plattform und die Dokumentation dazu findet man unter *„https://vaadin.com/flow"*.

Ein obligatorisches „Hello World": Hier hat sich im Vergleich zu Vaadin 8 einiges verändert, besser gesagt, vereinfacht. Bei einer Vaa-

```
```xml
<properties>
    <maven.compiler.release>10</maven.compiler.release>
</properties>
```
```

Listing 2

din- oder Flow-Anwendung wird mindestens ein Servlet benötigt. Dieser initiale Einstiegspunkt musste unter Vaadin 8 noch selber in Java geschrieben werden. Das kann man sich nun sparen.

Beginnen wir zuerst mit der Bereitstellung eines Servlet-Containers. Für dieses Beispiel dient das Apache-Projekt „meecrowave". In diesem Beispiel kommt das OpenJDK 10 zum Einsatz, was bedingt, dass das Attribut „release" dementsprechend in der „pom.xml" gesetzt wird *(siehe Listing 2)*. Die Abhängigkeiten, um Apache „meecrowave" zu verwenden, sind ebenfalls recht überschaubar *(siehe Listing 3)*.

Um den Servlet-Container nun zu starten, benötigen wir noch eine Klasse mit einer „Main"-Methode. Darin wird der Container in diesem Beispiel von Hand konfiguriert und gestartet *(siehe Listing 4)*.

Damit ist die Infrastruktur so weit vorhanden. Kommen wir nun zu den Abhängigkeiten, die wir für die Vaadin-Anwendung benötigen. Zuerst holen wir uns die Versions-Definitionen, die in der verwendeten Vaadin-Plattform-Version gültig sind *(siehe Listing 5)*.

Hier gibt es nun etwas Neues zu beachten. Seit der Version 10 wurde das Versionsschema geändert. Bis zu der Version 8 gab es immer eine führende Hauptversion. Die Version 9 wurde aufgrund der großen Unterschiede zu Version 8 einfach übersprungen und es geht demnach gleich mit der Version 10 weiter.

Dabei handelt es sich ja um die Version der Plattform. Die Plattform selbst ist ein Konglomerat von Versionen einzelner Komponenten. So hat der Button als „WebComponent" eine eigene Version. Diese WebComponent des Buttons wird dann in der dazugehörigen Vaadin-Flow-Version verwendet. Alles zusammen ist in der Version der Vaadin-Plattform zusammengeführt. Das hat nun folgende Punkte, die für ein langlebiges Projekt wichtig sind.

Als Erstes handelt es sich bei der Version 10 um eine LTS-Version. Das bedeutet, dass diese Version über zehn Jahre gepflegt wird. Dazu erscheint alle sechs Monate eine weitere Hauptversion der Plattform. Dabei handelt es sich jedoch um eine kurzlebige Version in Bezug auf den Support. In diesen Versionen gibt es Weiterentwicklungen, die nicht unbedingt auch in die LTS-Version zurückportiert werden. Daraus ergibt sich der nächste Punkt.

Bei Verwendung der Plattform in einer Version (LTS oder nicht, ist dabei nicht von Bedeutung) kann die Version einzelner Komponenten verändert werden. Es kann demnach in der Version 10 (LTS) die Version etwa des Buttons als WebComponent erhöht werden, weil in dieser neuen Version eine neue Funktion implementiert worden ist, die für das Projekt von Bedeutung ist.

Es muss also nicht die gesamte Plattform in der Version angepasst werden. Man kann sich die Version der Plattform daher so vorstellen, dass es sich um eine getestete Kombination von Versionen einzelner Komponenten handelt. Besondere Aufmerksamkeit in Bezug auf Stabilität und Wartung einschließlich Sicherheits-Patches er-

````
```xml
    <!--Infrastructure-->
    <dependency>
      <groupId>org.apache.meecrowave</groupId>
      <artifactId>meecrowave-core</artifactId>
      <version>${meecrowave.version}</version>
      <scope>compile</scope>
    </dependency>
```
````

Listing 3

````
```java
public class BasicTestUIRunner {
  private BasicTestUIRunner() {
  }

  public static void main(String[] args) {
    new Meecrowave(new Meecrowave.Builder() {
      {
//          randomHttpPort();
        setHttpPort(8080);
        setTomcatScanning(true);
        setTomcatAutoSetup(false);
        setHttp2(true);
      }
    })
        .bake()
        .await();
  }
}
```
````

Listing 4

````
```xml
  <dependencyManagement>
    <dependencies>
      <!--Vaadin -->
      <dependency>
        <groupId>com.vaadin</groupId>
        <artifactId>vaadin-bom</artifactId>
        <version>${vaadin.version}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
```
````

Listing 5

````
```xml
    <dependency>
      <groupId>com.vaadin</groupId>
      <artifactId>vaadin</artifactId>
    </dependency>

    <dependency>
      <groupId>com.vaadin</groupId>
      <artifactId>vaadin-lumo-theme</artifactId>
    </dependency>
```
````

Listing 6

```
```java
@Route("")
public class VaadinApp extends Composite<VerticalLayout> {

  private final Button btnClickMe = new Button("click me");
  private final Span lbClickCount = new Span("0");

  private int clickcount = 0;

  public VaadinApp() {
    btnClickMe.addClickListener(event -> lbClickCount.setText(valueOf(++ clickcount)));
    getContent().add(btnClickMe , lbClickCount);
  }
}
```
```

Listing 7

fahren die LTS-Versionen, da diese eben in dem Zeitraum von zehn Jahren gepflegt werden. Die nun verwendeten Abhängigkeiten in diesem Beispiel sind die beiden Hauptabhängigkeiten, die einem alle Open-Source-Bestandteile in dem Projekt zur Verfügung stellen *(siehe Listing 6)*.

Kommt es bei der Anwendung auf eine möglichst geringe Größe des Deployments an, kann man natürlich auch lediglich die jeweils benötigten Elemente definieren. Also zum Beispiel nur den Button sowie ein Label und das dann auch mit beziehungsweise ohne Java-API.

Die Implementierung

In Vaadin 8 war es noch notwendig, ein Servlet und eine Implementierung der „Basis UI"-Klasse selbst zu definieren. In der Realität sind diese allerdings meist immer gleich definiert; Anpassungen sind an dieser Stelle eher selten. Aus diesem Grunde wird mit Vaadin 10 nun auch eine Kombination aus Default-Implementierungen geliefert. Diese kommen zum Einsatz, wenn keine eigenen mit in den Klassenpfad gelegt werden.

Es kann also direkt mit der Definition der UI selbst begonnen werden. Neu ist auch die Definition der jeweiligen Route-Elemente. Als Erstes implementiert man eine Startseite, auf der ein Button und ein Label zu sehen sind. Dazu erzeugt man eine Klasse mit dem Namen „VaadinApp" und erbt von der Klasse „Composite<VerticalLayout>".

Bei der Klasse „Composite<T>" handelt es sich um einen Holder eines bestimmten Typs, der selber allerdings in dem zu erzeugenden DOM-Baum neutral ist. Nun stellt sich natürlich sofort die Frage nach dem „Warum?". Wenn zum Beispiel von der Klasse „VerticalLayout" geerbt wird, so ist die Komponente selbst ja auch wieder ein Layout. Allerdings ist es oft so, dass das Layout ja nur zur Strukturierung der benötigten Elemente verwendet werden soll: Die Komponente selbst ist jedoch kein Layout. Also wurde die Klasse „Composite" eingeführt. Ein ähnliches Konstrukt gibt es auch in zwei Varianten in Vaadin 8. Hier wurde demnach aufgeräumt und ein neuer, einheitlicher Weg definiert.

Innerhalb der Klasse „VaadinApp" hat man dann die Instanz des Typs, in diesem Fall die Klasse „Div", mit der Methode „getContent()" als Zugriff. Diese Instanz kann man dann verwenden, um die Kind-Elemente hinzuzufügen. *Listing 7* zeigt genau das mit einem Button und einem Span.

Das Erzeugen der Elemente erfolgt wie ein ganz normales Attribut der Klasse. Die beiden Elemente werden dann dem Layout, das die Root-Komponente darstellt, mit „getContent().add(btnClickMe , lbClickCount);" hinzugefügt.

Die Annotation der Klasse „@Route("")" definiert diese grafische Komponente als Root-Komponente. Die Attribut-Elemente der Annotation enthalten noch mehr Elemente: „@Route(value = "", layout = UI.class, absolute = false)". Der Default-Wert gibt den Navigationsteil an, unter dem diese Komponente zu erreichen sein soll. Zusammen mit dem Attribut „absolute" wird der finale Pfad bestimmt.

Als Standartwert „absulute=false" wird der Teil, der durch das Attribut „value = """ angegeben ist, als relatives Pfad-Element ausgewertet. Wenn man nun die „Main"-Methode der zuvor erzeugten Klasse „VaadinApp" aufruft, kann man nachfolgend unter der URL *„http://localhost:8080"* die Anwendung ausprobieren *(siehe Abbildung 2)*.

Das Layout

Das Layout einer Anwendung lässt sich auf viele Arten definieren und anpassen. Auch bei Vaadin ist es möglich, zum Beispiel durch CSS Anpassungen vorzunehmen. Hier wird allerdings der Weg beschrieben, der mithilfe programmatischer Layouts gegangen werden kann. Wer mehr zum Thema „CSS und Styling" lesen möchte, dem sind die entsprechenden Seiten unter *„vaadin.com"* empfohlen. Ein Einstiegspunkt könnte *„https://vaadin.com/docs/v11/flow/importing-dependencies/tutorial-include-css.html"* sein.

In der Annotation „@Route" ist auch das Attribut „layout" enthalten. Hier kann man eine Klasse angeben. Die angegebene Klasse selbst kann man sich nun wie einen Behälter oder besser gesagt wie einen Rahmen um die später anzuzeigende Komponente vorstellen.

```java
public class MainLayout extends Composite<Div> implements RouterLayout {

  private Div content = new Div();

  public MainLayout() {
    final VerticalLayout layout = new VerticalLayout(
        new Span("from MainLayout top") ,
        content,
        new Span("from MainLayout bottom")
    );
    getContent().add(layout);
  }

  //SNIPP...
}
```

Listing 8

```java
public class MainLayout extends Composite<Div> implements RouterLayout {

  private Div content = new Div();

  //SNIPP - constructor

  @Override
  public void showRouterLayoutContent(HasElement hasElement) {
    Objects.requireNonNull(hasElement);
    Objects.requireNonNull(hasElement.getElement());
    content.removeAll();
    content.getElement().appendChild(hasElement.getElement());
  }
}
```

Listing 9

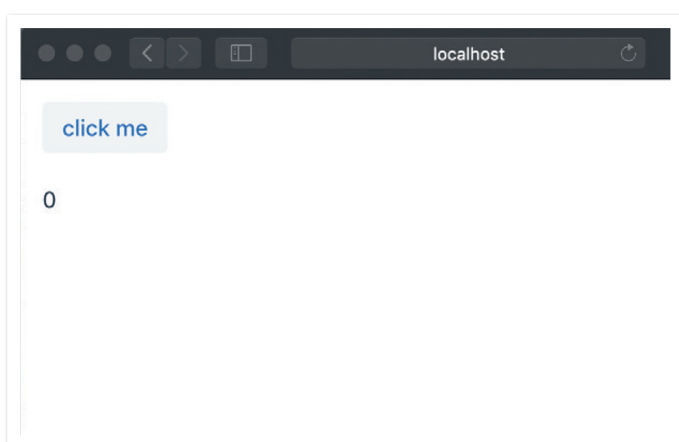

Abbildung 2: Click me

Die Klasse „MainLayout" im Beispielprojekt stellt das Hauptgerüst der Anwendung dar. Das wird hier stark vereinfacht dargestellt, indem oben und unten am Browser-Fenster jeweils ein Text angezeigt wird. Zwischen diesen beiden Texten soll nun der Rest der Anwendung dargestellt werden. Als Container für diesen Inhalt ist eine Instanz der Klasse „Div" bereitgestellt *(siehe Listing 8)*.

Was nun noch fehlt, ist der Mechanismus, um eine Komponente in das vorgehaltene „Div" zu setzen. Zum einen benötigen wir den programmatischen Teil im Layout selbst. Dazu wird die Methode „show RouterLayoutContent(HasElement hasElement)" aus dem Interface „RouterLayout" überschrieben. In dieser Methode wird das übergebene Element in den dafür vorgesehenen Container eingesetzt. Zuvor werden eventuell vorhandene Elemente entfernt *(siehe Listing 9)*.

Um nun dieses erste Basis-Layout zu verwenden, wird die Klasse „VaadinApp" mit dem neu erzeugten Layout versehen. Das erfolgt dadurch, dass nun die Klasse „MainLayout" als Wert dem Attribut „layout=MainLayout.class" innerhalb der Annotation „@Route" übergeben wird.

Wenn wir nun die Anwendung starten, erhalten wir eine Exception. Der Grund dafür ist in der Fehlernachricht zu lesen. Wenn man mit einem Layout arbeitet, so gehören die Annotationen „@Theme(..)" an die Klasse, die das Layout definiert; in diesem Fall an die Klasse „MainLayout" *(siehe Abbildung 3)*.

Kommen wir nun zu dem Punkt, in dem das Layout aus verschiedenen Elementen zusammengebaut wird. Zum Beispiel soll ein

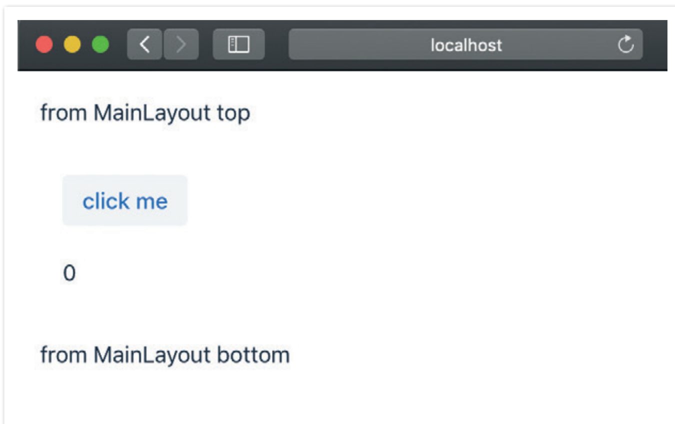

Abbildung 3: MainLayout

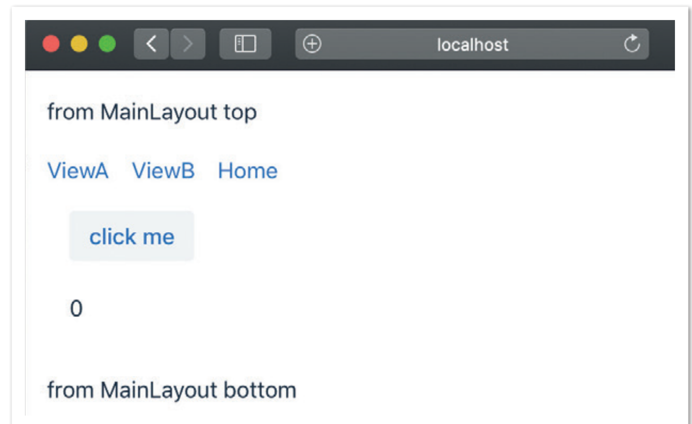

Abbildung 4: Das fertige Layout

````
```java
@ParentLayout(value = MainLayout.class)
public class LayoutWithMenuBar extends Composite<Div> implements RouterLayout {

  public LayoutWithMenuBar() {
    final HorizontalLayout layout = new HorizontalLayout(
        new RouterLink("ViewA" , ViewA.class) ,
        new RouterLink("ViewB" , ViewB.class) ,
        new RouterLink("Home" , VaadinApp.class)
    );
    getContent().add(layout);
  }
}
```
````

Listing 10

````
```java
@Route(value = "ViewA", layout = LayoutWithMenuBar.class)
public class ViewA extends Composite<Div> {

  public ViewA() {
    getContent().add(new Span("ViewA"));
  }
}

@Route(value = "ViewB", layout = LayoutWithMenuBar.class)
public class ViewB extends Composite<Div> {

  public ViewB() {
    getContent().add(new Span("ViewB"));
  }
}
```
````

Listing 11

Menü allgemein definiert werden, auch dieses ist Teil des Layouts. Hier kommen wir zu dem Begriff „ParentLayout". Mit der Annotation „@ParentLayout(..)" kann ein übergeordnetes Layout angegeben werden.

Das neue Layout, das wir für dieses Beispiel erzeugen wollen, erhält den Namen „LayoutWithMenuBar". Dieses erbt nicht von unserem „MainLayout", sondern erweitert dieses. Die Beziehung zwischen diesen beiden Layout-Ebenen wird mithilfe der Annotation „@ParentLayout(value = MainLayout.class)" an der Klasse „LayoutWithMenuBar" hergestellt *(siehe Listing 10)*. Bei den angegebenen Klassen „ViewA" und „ViewB" handelt es sich lediglich um Platzhalter für theoretische Navigationsziele innerhalb der Anwendung *(siehe Listing 11)*.

```java
@ParentLayout(value = MainLayout.class)
public class LayoutWithMenuBar extends Composite<Div> implements RouterLayout{

  private final Div content = new Div();

  private final HorizontalLayout menuBar = new HorizontalLayout(
      new RouterLink("ViewA" , ViewA.class) ,
      new RouterLink("ViewB" , ViewB.class) ,
      new RouterLink("Home" , VaadinApp.class)
  );

  public LayoutWithMenuBar() {
    getContent().add(new VerticalLayout(menuBar, content));
  }

  @Override
  public void showRouterLayoutContent(HasElement hasElement) {
    Objects.requireNonNull(hasElement);
    Objects.requireNonNull(hasElement.getElement());
    content.removeAll();
    content.getElement().appendChild(hasElement.getElement());
  }
}
```

Listing 12

Alle Views werden mit demselben gerade erweiterten Layout versehen. Wird die Anwendung nun gestartet, so ergibt sich das Bild in *Abbildung 4*. Wie wir sehen, sind die Elemente wie vorgesehen und ineinander geschachtelt. Wir können nun in dem angebotenen Menü navigieren und erhalten immer dasselbe Layout, allerdings mit wechselndem Inhalt, der durch die jeweilige View gesetzt worden ist.

Die Lösung ist allerdings nur zum Teil korrekt. Wenn man die erzeugten HTML-Elemente betrachtet, sieht man, dass die Menü-Elemente und die Inhalte der Arbeitsfläche zusammen in einem „Div" enthalten sind. Allerdings ist es von der Logik wohl eher so gemeint, dass die jeweiligen Arbeitsflächen innerhalb des letzten Layout-Elements enthalten sind.

Um hier genau das gleiche Verhalten zu realisieren, muss auch in dem erweiterten Layout die Methode „showRouterLayoutContent" implementiert werden. Damit ist nun sichergestellt, dass sich die Teil-Layouts neutral verhalten und keine implizierten Anforderungen enthalten sind *(siehe Listing 12)*.

Fazit

Das soll als kleine Einführung in Vaadin10 genügen; wer mehr wissen möchte, sollte einen Blick in die Vaadin-Online-Dokumentation werfen. Dort gibt es neben der Übersicht über alle verfügbaren Komponenten auch ein ausführliches Online-Handbuch.

Hinweis: Die Sourcen zu diesem Artikel sind auf GitHub unter *„https://github.com/Java-Publications/vaadin-v10-java-aktuell-intro"* zu finden.

Sven Ruppert
sven.ruppert@gmail.com

Sven Ruppert programmiert bereit seit dem Jahr 1996 in Java. Als Oracle Developer Champion und Developer Advocate bei Vaadin hilft er weltweit Entwicklern beim Wachstum ihres Geschäfts. In seiner Freizeit spricht Sven Ruppert auf internationalen und nationalen Konferenzen und schreibt für IT-Magazine sowie für Tech-Portale.