www.javaspektrum.de

Ausgabe 1/2023
D € 12,90 · A € 13,90 · CHF 20,20

4 194156 112901 01

JavaSPEKTRUM

Magazin für professionelle Entwicklung und digitale Transformation

Power of JavaScript –
Frameworks & Bibliotheken

React – Eine praktische Einführung

Java-Full-Stack-Entwicklung mit Hilla

Interview

Dr. Lutz Seidenfaden, CIO von MTU Aero Engines, über die Herausforderungen der Digitalisierung und Vereinfachung der Prozesse

Fachthemen

Von Spring Boot nach Quarkus – in der Cloud macht man das so

Nachhaltigkeit im Software-Engineering – Teil 5: Klimakompensation

Power of JavaScript – Frameworks & Bibliotheken

G30325

Sonderdruck aus
JavaSPEKTRUM 1/2023

TypeScript und Java integriert

Java-Full-Stack-Entwicklung mit Hilla

Simon Martinelli

Das Framework Hilla verspricht eine Effizienzsteigerung bei der Entwicklung von Webapplikationen durch die Integration eines Spring Boot-Java-Backend mit einem reaktiven TypeScript-Frontend. Benutzeroberflächen werden mit dem Lit-Framework und Webkomponenten erstellt, und man muss sich nicht um die REST-API kümmern. Hilla generiert die REST-API und den Zugriffscode für den Client. Das Backend ist standardmäßig sicher und vollständig zustandslos. Hält Hilla, was es verspricht?

Das Hilla-Framework [Hilla] wird von der finnischen Firma Vaadin, die das gleichnamig Java-Web-Framework pflegt, entwickelt. Hilla ist der finnische Name für Moltebeere, auch Sumpfbeere genannt. Im Gegensatz zu Vaadin, das einen reinen Java-Ansatz verfolgt, ist Hilla ein klassisches Single-Page Application (SPA)-Framework, das aber ebenfalls die Full-Stack-Entwicklung im Fokus hat. Das heißt, der Client wird in TypeScript entwickelt. Im Frontend wird das Lit-Framework [Lit] eingesetzt und im Backend kommt aktuell ausschließlich Spring Boot zum Einsatz, es wird an der Unterstützung für andere Java-Frameworks gearbeitet.

Ein Hilla-Projekt ist ein reines Maven-Projekt. Unter der Haube verwendet das Hilla-Maven-Plug-in npm und Vite für den Frontend-Build. Im Gegensatz zu herkömmlicher Frontend-Entwicklung muss man sich aber nicht um die Konfiguration und Ausführung dieser Werkzeuge kümmern, was insbesondere für Java-Entwickler den Einstieg in die Frontend-Entwicklung stark vereinfacht.

Lit

Lit ist der Nachfolger der bekannten Polymer-Bibliothek [Polymer] und wird für die einfache Entwicklung von Webkomponenten [Web-Components] verwendet. Mit Lit lassen sich sogenannte Custom Components, also Erweiterungen der HTML-Sprache, entwickeln. Die Templates sind deklarativ im TypeScript-Code eingebunden und auch CSS, das nur im Kontext der Webkomponente gültig ist, lässt sich hinzufügen. Eigenschaften der Webkomponente sind reaktiv und führen bei Änderungen zu einem automatischen Neurendern.

Listing 1 zeigt eine einfache Webkomponente. Wichtig ist der Name im `@customElement`-Decorator, der einen Bindestrich enthalten muss, um von einem Standard-HTML-Element unterschieden zu werden. Der Decorator `@property` macht den String `name` zu einer reaktiven Property, welche erstens von außen auf die Komponente gesetzt werden kann und zweitens beim Ändern dazu führt, dass die Komponente neu gerendert wird. Und in der Methode `render()` wird die Vorlage für die Webkomponente erzeugt. Im erzeugten DOM findet man die Komponente wie in Listing 2.

```
@customElement('simple-greeting')
export class SimpleGreeting extends LitElement {
  @property()
  name?: string = 'World';

  render() {
    return html`<p>Hello, ${this.name}!</p>`;
  }
}
```

Listing 1: Komponente mit Lit

```
<body>
  <simple-greeting name="World"></simple-greeting>
</body>
```

Listing 2: Gerenderte Webkomponente

Simon Martinelli ist Inhaber der 72 Services GmbH und seit 27 Jahren als Softwarearchitekt, -entwickler, Berater und Trainer vor allem im Java-Enterprise-Umfeld unterwegs. Aktuell gilt sein Interesse der Effizienzsteigerung bei der Full-Stack-Entwicklung mit Java. Aufgrund seines Engagements im Vaadin/Hilla-Umfeld wurde ihm der Vaadin Community Award verliehen. Als Dozent an der Berner Fachhochschule in den Bereichen Architektur und Design von verteilten Applikationen und Persistenztechnologien kann er seine Interessen vertiefen und sein Know-how weitergeben.
Twitter: @simas_ch, E-Mail: simon@martinelli.ch,
Web: https://martinelli.ch

Endpoints

Auf Backendseite verwendet Hilla sogenannte Endpoints. Ein Endpoint ist eine Spring Bean, die mit der Annotation `@Endpoint` annotiert wird. Davon erzeugt Hilla eine REST-API inkl. TypeScript-Code für den Zugriff auf der Client-Seite.

Was in Listing 3 als Erstes auffällt, ist die Annotation `@AnonymousAllowed`. Diese ist nötig, um unangemeldet auf die API zugreifen zu können, da bei Hilla alle Endpoints standardmäßig vor Zugriffen geschützt sind. Auch zu beachten ist die Annotation `@Nonull`. Da TypeScript strikter mit Null umgeht als Java, kann damit dem TypeScript-Generator mitgeteilt werden, dass sowohl der Parameter als auch der Rückgabewert nie Null enthalten.

Listing 4 zeigt den generierten TypeScript-Code, der im Frontend verwendet werden kann. Sollte sich am Endpoint oder an den Parameter- oder Rückgabetypen etwas ändern, so wird der Code neu generiert und auf der Clientseite wird dementsprechend ein Fehler gemeldet. Damit werden Fehler bei der Verwendung der API bereits zur Entwicklungszeit erkannt.

Die Beispielapplikation

Die Applikation soll eine Tabelle mit Personendaten anzeigen, deren Einträge mittels eines Formulars bearbeitet werden können. Die Personendaten werden mithilfe von JPA in der Datenbank abgespeichert. Abbildung 1 zeigt, wie das Endresultat aussehen wird. Der Beispielcode ist auf GitHub publiziert [Code].

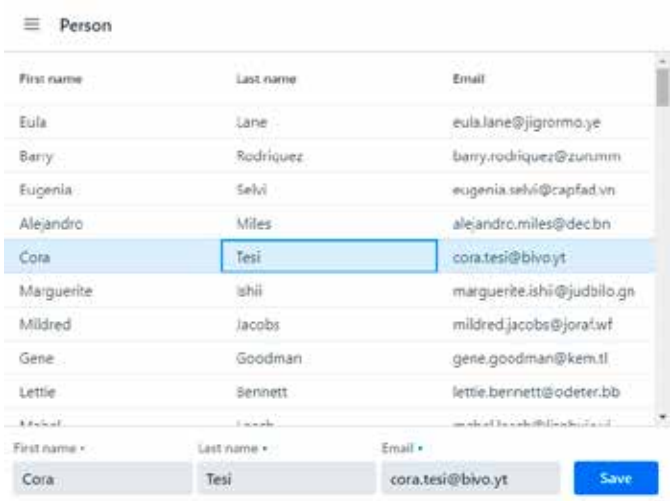

Abb. 1: Grid mit Formular

```
@Endpoint
@AnonymousAllowed
public class HelloWorldEndpoint {
  @Nonnull
  public String sayHello(@Nonnull String name) {
    if (name.isEmpty()) {
      return "Hello stranger";
    } else {
      return "Hello " + name;
    }
  }
}
```

Listing 3: Endpoint

```
function _sayHello(name: string): Promise<string> {
  return client.call('HelloWorldEndpoint', 'sayHello', {name});
}
export { _sayHello as sayHello };
```

Listing 4: Generierter TypeScript-Code

```
npx @hilla/cli init hilla-app
```

Listing 5: CLI

CLI

Um eine Hilla-Applikation zu erstellen, muss zuerst NodeJS [NodeJS] in der Version 16.14 oder neuer installiert werden. Danach kann mit npx die Vaadin CLI verwendet werden, um ein neues Projekt zu erstellen. Die CLI generiert eine vollständige Hilla-Applikation mit einer Hello-World-View und dem HelloWorld-Endpoint aus Listing 5.

Backend

Als Erstes wird eine Entität mit dem Namen `Person` hinzugefügt. Das Beispiel verwende JPA, um die Daten in einer H2-Datenbank zu speichern. Wie in Listing 6 zu sehen, werden Bean Validation-Annotationen verwendet. Auch diese werden vom Hilla-Generator berücksichtigt. Wenn die `Person`-Entity im Client in einem Formular verwendet wird, werden die Eingaben entsprechend den Annotationen validiert (s. Abb. 2).

Als Nächsten wird der Endpoint erstellt, um die Daten der Personen zu lesen und zu speichern. Das verwendete `PersonRepository` in Listing 7 erweitert das Spring Data JPA JpaRepository Interface.

Frontend

Auf der Client-Seite wird eine View für die Darstellung der Personendaten benötigt, für die Bearbeitung dieser Daten wird ein Formular erstellt. Zunächst zum *Anzeigen der Personen*.

Auf der Client-Seite wird eine View für die Darstellung der Personendaten benötigt, welche ein Vaadin-Grid verwendet. Alle Komponenten von Vaadin sind Webkomponenten und können deshalb sehr einfach mit Lit verwendet werden. Das Vaadin-Grid bietet Paging, Sortierung und viele weitere Funktionen, die es sehr einfach machen, Daten in Tabellenform darzustellen.

In der Methode `connectedCallback`, die aufgerufen wird, wenn die Webkomponente dem DOM hinzugefügt wird, werden die Personen vom Endpoint gelesen (s. Listing 9). Die Personen werden der Eigenschaft `items` in dem Vaadin-Grid hinzugefügt, und die Eigenschaft „path“ bei den Spalten definiert den Pfad zur Eigenschaft

First name •	Last name •	Email •
must not be blank	must not be blank	must be well-formed email address

Abb. 2: Validierung

```
@Entity
public class Person {
  @Id @GeneratedValue
  private Long id;

  @NotBlank
  private String firstName;

  @NotBlank
  private String lastName;

  @Email @NotBlank
  private String email;
  ...
}
```

Listing 6: Person Entity

```
@Endpoint
@AnonymousAllowed
public class PersonEndpoint {
  @Autowired
  private PersonRepository personRepository;

  @Nonnull
  public List<@Nonnull Person> findAll() {
    return personRepository.findAll();
  }

  public void save(@Nonnull Person person) {
    this.personRepository.save(person);
  }
}
```

Listing 7: Person Endpoint

```
public interface PersonRepository
    extends JpaRepository<Person, Integer> {
}
```

Listing 8: Person Repository

```
@customElement('person-view')
export class PersonView extends View {
  @state()
  people: Person[] = [];

  async connectedCallback() {
    super.connectedCallback();
    this.people = await PersonEndpoint.findAll();
  }

  render() {
    return html`
      <vaadin-grid .items=${this.people} style="height: 100%">
        <vaadin-grid-column path=
          "firstName"></vaadin-grid-column>
        <vaadin-grid-column path=
          "lastName"></vaadin-grid-column>
        <vaadin-grid-column path="email"></vaadin-grid-column>
      </vaadin-grid>
    `;
  }
}
```

Listing 9: Person View

unserer Person. Der Einfachheit halber verwendet dieses Beispiel kein Paging. Sollte die Tabelle eine größere Anzahl Datensätze enthalten, so sollte unbedingt Paging verwendet werden, um eine Teilmenge der Daten zu laden. Hilla bietet dazu einen DataProvider an, der die Informationen über die aktuell angezeigte Seite, die Seitengröße und die gewählte Sortierung zur Verfügung stellt und beim Blättern die Daten jeweils seitenweise vom Endpoint anfordert. Ein ausführliches Code-Beispiel ist unter [MasterDetail] zu finden.

Nun zur *Bearbeitung der Person*. Für die Bearbeitung der Personendaten wird ein Formular erstellt. Dazu werden, wie in Listing 10 zu sehen, Vaadin-Webkomponenten verwendet. Um eine Entität `Person` an die Komponenten zu binden, stellt Hilla einen Binder zur Verfügung (s. Listing 11). Der Binder verwendet die generierte Klasse `PersonModel`, die Zusatzinformationen zur Entität `Person`, wie Validierung oder Typ, enthält.

```
<vaadin-form-layout>
  <vaadin-text-field
    label="First name"
    ${field(this.binder.model.firstName)}
  ></vaadin-text-field>
  <vaadin-text-field
    label="Last name"
    ${field(this.binder.model.lastName)}
  ></vaadin-text-field>
  <vaadin-text-field
    label="Email"
    ${field(this.binder.model.email)}
  ></vaadin-text-field>
</vaadin-form-layout>
<vaadin-button @click=${this.save}>Save</vaadin-button>
```

Listing 10: Formular

```
private binder = new Binder<Person, PersonModel>(this, PersonModel);
```

Listing 11: Binder

```
private async save() {
  await this.binder.submitTo(PersonEndpoint.save);
  this.people = await PersonEndpoint.findAll();
}
```

Listing 12: Save-Methode

```
<vaadin-grid
  .items=${this.people}
  @active-item-changed=${this.itemSelected}
  .selectedItems=${[this.selectedPerson]}>
```

Listing 13: Grid-Selektion

```
private async itemSelected(event: CustomEvent) {
  this.selectedPerson = event.detail.value as Person;
  this.binder.read(this.selectedPerson);
}
```

Listing 14: itemSelected-Methode

Damit die veränderte Entität Person gespeichert werden kann, erweitern wir den `PersonEndpoint` um die Methode `save`. Diese Methode kann direkt dem Binder übergeben werden. Dazu wird das Click-Event an den Button gebunden (s. Listing 10) und die Methode `save` aufgerufen. Nach dem Speichern werden die Personendaten neu geladen und dadurch wird das Grid aktualisiert (s. Listing 12).

Was nun noch fehlt, ist das Übergeben der im Grid selektierten Person an den Binder. Dazu kann das Event `active-item-changed` verwendet werden (s. Listing 13). Auch muss dem Grid mitgeteilt werden, welche `Person` selektiert ist, dazu dient die Eigenschaft `selectedItems`. Jetzt muss in der `itemSelected`-Methode in Listing 14 nur noch die selektierte `Person` aus dem Event gelesen und dem Binder übergeben werden. Damit wird das Formular befüllt.

Routing

Wenn die Applikation mehr als eine View umfasst, braucht es eine Möglichkeit, zwischen den Views zu navigieren. Dazu verwendet

```
import {Route} from '@vaadin/router';
import './views/helloworld/hello-world-view';
import './views/main-layout';

export type ViewRoute = Route & {
  title?: string;
  icon?: string;
  children?: ViewRoute[];
};

export const views: ViewRoute[] = [
  {
    path: '',
    component: 'hello-world-view',
    icon: '',
    title: '',
  },
  {
    path: 'hello-world',
    component: 'hello-world-view',
    icon: 'la la-globe',
    title: 'Hello World',
  },
  {
    path: 'master-detail',
    component: 'master-detail-view',
    icon: 'la la-columns',
    title: 'Master-Detail',
    action: async (_context, _command) => {
      await import('./views/masterdetail/master-detail-view');
      return;
    },
  },
];
export const routes: ViewRoute[] = [
  {
    path: '',
    component: 'main-layout',
    children: [...views],
  },
];
```

Listing 15: Router-Konfiguration

Hilla den Vaadin-Router (s. Listing 15). Zuerst wird die View importiert, die beim Starten der Applikation angezeigt wird, hier die `hello-world-view`. Diese wird dann einmal auf den Root-Pfad und einmal auf den Pfad `hello-world` gemappt. Die andere View, im Beispiel `master-detail-view`, wird lazy geladen, damit diese nur geladen wird, wenn der Benutzer auch tatsächlich darauf navigiert. Als letztes wird für die Views noch ein Layout definiert, das Elemente wie Kopf- und Fußzeile sowie die Navigationskomponente enthält.

Deployment in Produktion

Standardmäßig sind Hilla-Anwendungen so konfiguriert, dass sie im Entwicklungsmodus ausgeführt werden. Dies erfordert etwas mehr Speicher und CPU-Leistung, ermöglicht aber ein einfacheres Debugging. Für das Deployment muss die Applikation im Produktionsmodus gebaut werden. Der Hauptunterschied zwischen Entwicklungs- und Produktionsmodus besteht darin, dass Hilla im Entwicklungsmodus Vite verwendet, um JavaScript-Dateien an den Browser zu liefern, anstatt an den Java-Server, auf dem die Anwendung ausgeführt wird. Wenn eine JavaScript- oder CSS-Datei geändert wird, werden die Änderungen übernommen und automatisch bereitgestellt. Im Produktionsmodus jedoch ist es effizienter, JavaScript- und CSS-Dateien einmal während des Bauens vorzubereiten und einen Server alle Anfragen bedienen zu lassen. Gleichzeitig können die Client-Ressourcen optimiert und minimiert werden, um die Belastung von Netzwerk und Browser noch weiter zu reduzieren.

```
<profiles>
  <profile>
    <id>production</id>
    <build>
      <plugins>
        <plugin>
          <groupId>dev.hilla</groupId>
          <artifactId>hilla-maven-plugin</artifactId>
          <version>${hilla.version}</version>
          <executions>
            <execution>
              <goals>
                <goal>build-frontend</goal>
              </goals>
              <phase>compile</phase>
            </execution>
          </executions>
          <configuration>
            <productionMode>true</productionMode>
          </configuration>
        </plugin>
      </plugins>
    </build>
  </profile>
</profiles>
```

Listing 16: Maven-Plug-in

```
./mvnw package -Pproduction
```

Listing 17: Production Build

Die Datei pom.xml in einem Hilla-Projekt verwendet ein Profil mit der Konfiguration des Vaadin-Plug-ins, um einen Build im Produktionsmodus zu erstellen (Listing 16). Um einen Produktions-Build zu erstellen, können Sie Maven wie in Listing 17 aufrufen. Dadurch wird eine JAR-Datei mit allen Abhängigkeiten und transpilierten Frontend-Ressourcen erstellt, die danach deployt werden kann.

Fazit

Durch die Generierung des Zugriffscodes auf die Endpoints und der Modellklassen ist die Integration von Frontend und Backend viel einfacher als bei herkömmlicher Single-Page Application-Entwicklung. Die Vaadin-Webkomponenten wie das Grid sind bei der Entwicklung von datenlastigen Applikationen äußerst hilfreich. Der Binder, insbesondere in Kombination mit Bean Validation, macht es sehr einfach, Formulare zu erstellen, und reduziert den Code auf ein Minimum. Da sich der Entwickler nicht um den Frontend-Build und die -Tools kümmern muss, eignet sich Hilla auch sehr gut für Java-Entwickler.

Im Artikel wurde nur auf die wichtigsten Aspekte von Hilla eingegangen. Es fehlen noch einige Bestandteile, wie Styling und Theming, Security, Lokalisierung, Fehlerbehandlung oder applikationsweite Zustandsverwaltung, um eine voll ausgebaute Applikation zu erstellen. Diese und viele weitere Themen behandelt die offizielle Dokumentation [Hilla].

Literatur und Links

[Code] https://github.com/simasch/hilla-javaspektrum
[Hilla] https://hilla.dev/
[Lit] https://lit.dev/
[MasterDetail] https://github.com/simasch/hilla-master-detail-with-filter
[NodeJS] https://nodejs.org/en/
[Polymer] https://polymer-library.polymer-project.org/
[WebComponents] https://developer.mozilla.org/de/docs/Web/Web_Components