Web Frameworks: GWT

Roland Hediger

2. Februar 2014

Inhaltsverzeichnis

Teil I.

Theorie

1. Einführung

- Anfang 60er Jahre ersten Tastaturen und Monochrommonitore.
- Terminal Mainframe Systeme
- 70er Jahre PCs : Fat Clients
- Software Unterhaltung auf verscheidene Clients war problematisch.
- Zentral Installationspunkt nötig
- Web Applikationen aber mit viele Roundtrips.
- Letzten jharzent versucht teile der Applikationslogik wieder zum Benutzer auszulagern.
- Mit guter GUI und Server Verbindungsunterbruch.
- 90er : Quasi Standardsprache im Browser : Javascript. Vorraussetzungen für RIA.

1.1. RIA Interaktionsmodell

AJAX Asynchronous Javascript and XML aber ist mehr zi einem Befriff für ein Architekturkonzept geworden statt sich auf einzelnen Technologieen zu konzentrieren. Statt Javascript kann auch Actionscript von Adobe zum Einsatz kommen. Anstelle von XML kann man auch JSON übertragen oder Binär. Javascript hat den Vorteil das es keine Plugins braucht.

1.1.1. Klassisch

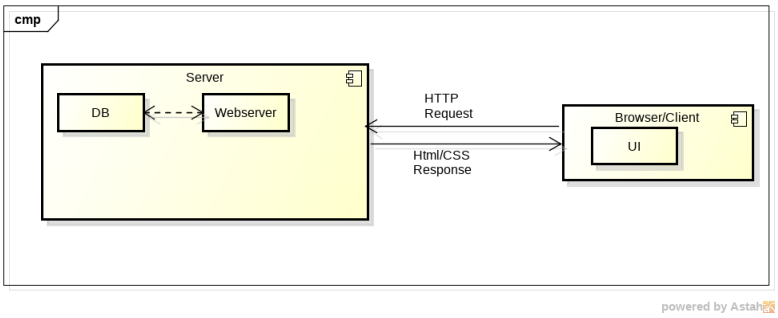


Abbildung 1.1.: figure

1.1.2. RIA

1.2. RIA Technologieen

Plugin Basiert Verschwenden Mittelfristig:

- Adobe Flex, Java Applets und JavaFX Microsoft Silverlight.

AJAX Frameworks Javscript Bibliotheken mit methoden die das Leben einfacher machen :

- Angular JS, Dojo, Enyo, JQuery, Sencha/ExtJS.

Handgeschriebenes Javascript Immer noch sehr weit verbereitet. Normalle Web App mit Javascript aufgepeppt. Argumente gegen der Einsatz :

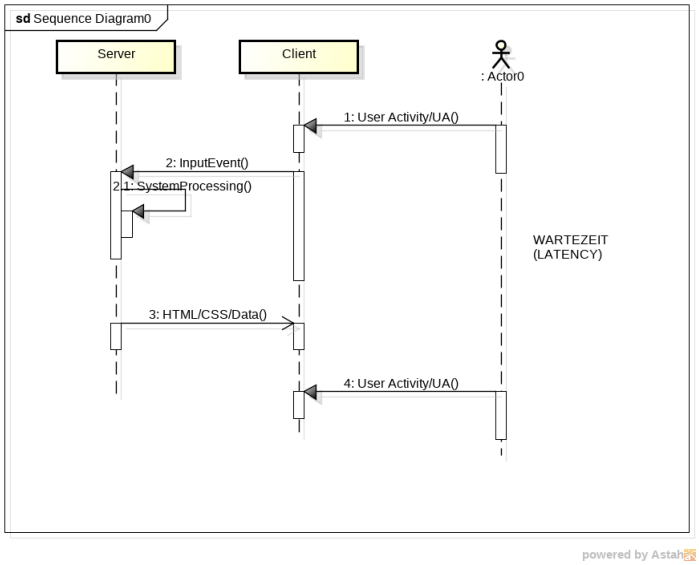


Abbildung 1.2.: figure

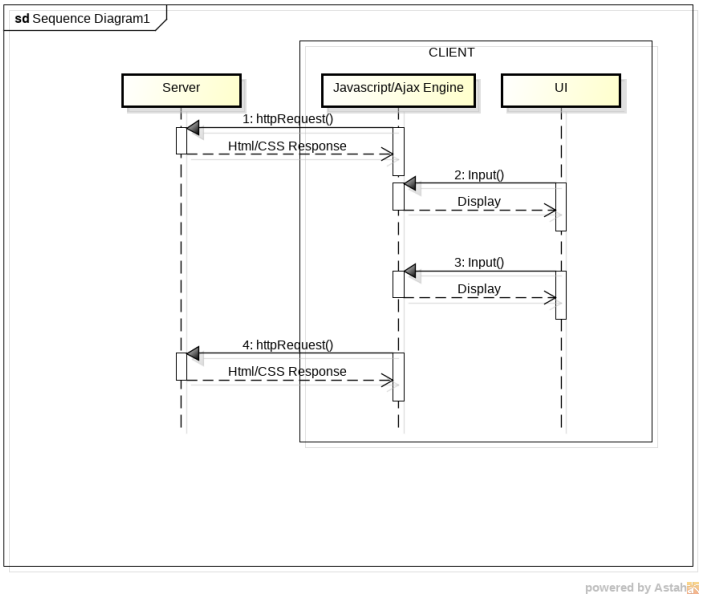


Abbildung 1.3.: figure

Javascript ist nicht Java Eigene Programmiersprache mit anderen konzepte als Java. Kein Typsystem, Interpretierte Sprache - viele Fehler manefestieren sich nur zur Laufzeit. Keine Klassen oder Vererbungshirachieen. Selbst von Hand implementieren. Objektorientiert - Hashtabelle die alles mögliche für Felder drin haben kann auch Funktionen als Felder. *Funktionen sind auch Objekte* Es ist möglich Funktionen als Parameter zu übergeben oder als Rcukgabewerte zu erhalten. Ist doch **eine Starke genenüber Java**

Browser Quirks Alle Browser implementieren Javascript ein bisschen anders weil es kein verbindlichen Standard gibt.

Zu viele Bibliotheken Viele Bibliotheken um das Leben einfacher zu machen. Viel zu lernen und schwierig den Überblick über alles zu behalten vorallem wenn man verschiedene Libs gemischt miteinander benutzt.

Bibliotheken helfen nur zum Teil Die konzentrieren sich nur auf gewisse Aspekte. Richtiges Entwicklen braucht Kenntnisse in mehreren Programmiersprachen und mehreren Programmiermodellen.

Komplett Generierter Code(GWT)

- Java als einzige Programmiersprache:

  Client-seitiger Code wird von Java nach Javascript transcompiliert.

GUIs können aus Javacode erstellt werden (fraglich ob das ein Vorteil ist?), aber auch mittels XML-Dateien spezifiziert und in den Code eingebunden werden.

- Das herkömmliche Java-Tooling kann weiterhin verwendet werden:

  JUnit, Emma, Mockito, Logging ...

  Ein grosser Teil der Standartbibliotheken stehen zur Verfügung.
- Die Kommunikation mit Servern: http-Requests werden in sogenannte Remote Procedure Calls (RPC) umgewandelt. Das sind Methodenaufrufe mit typisierten Parametern in Java!
- GWT abstrahiert den Browser und liefert Libraries mit, die den Code auf allen gängigen Browsern lauffähig machen.

1.3. Google Web Toolkit

Nicht indexierbar durch Search Engines – Dies ist ein allgemeines Problem vieler RIA- Anwendungen. Die Applikation verwendet nur eine einzige HTML-Seite. Das GUI wird durch Manipulationen des DOM-Trees der Seite verändert. Search Engines sehen nur den statischen Teil der Seite und nicht den dynamischen, durch Javascript verwalteten Teil.

Alles oder Nichts – Falls JavaScript im Browser ausgeschaltet wurde funktioniert die ganze Applikation nicht mehr (ausser einer einfachen Anzeige JavaScript zu aktivieren). Handgeschriebene RIAs können oft noch eine Grundfunktionalität ohne Javascript- Unterstützung anbieten. Diese Eigenschaft wird „graceful degradation“ genannt. GWT ist nicht gracefully degradable!

1.3.1. Java nach JavaScript Compiler

Dies ist das Herzstück von GWT und auch dessen grösster Vorteil. Der Compiler übersetzt ein erstaunlich grosses Subset von Java 5. Natürlich gibt es einige Einschränkungen – sei es weil JavaScript (JS) keine Unterstützung bietet, oder weil gewisse Features nicht (effizient) abgebildet werden können. Darunter fallen folgende Punkte:

- Reflection
- Multithreading und Synchronisation (JS Interpreter sind singlethreaded)
- Finalization
- strictfp, allg.: Präzisionsverluste möglich
- Typen: Zwar sind byte, char, short, int, long, float und double unterstützt, JS kennt aber nur Numeric, Boolean und String. Rundungs- und Abbildungsfehler sind zu erwarten.

1.3.2. Java Emulation

Es reicht nicht aus nur einen Compiler anzubieten. Das gesamte Java Runtime Environment (JRE) muss auf JavaScript abgebildet werden. Dies bedeutet, dass möglichst viele der Klassen aus der JRE nach JavaScript portiert werden müssen. Die sogenannte JRE Emulation Library bietet genau diese Portierung an. Eine ausführliche Liste der portierten Klassen und Methoden findet sich unter http://www.gwtproject.org/doc/latest/RefJreEmulation.html Nebst den Portierungen gibt es auch einige wenige alternative Klassen, die eine ähnliche, leicht angepasste Funktionalität wie ihre Originale anbieten:

```
http://www.gwtproject.org/doc/latest/DevGuideCodingBasicsCompatibility.html#similar
```

Zuletzt noch eine Warnung: Es können nicht beliebig Klassen oder Bibliotheken verwendet werden, da sie nicht als Java-Bytecode sondern in JavaScript vorliegen müssen. Selbst wenn der Sourcecode einer Library vorhanden ist, ist eine Compilation nach JavaScript nicht immer möglich!

1.3.3. UI Library

Nebst den JRE Libraries gibt es in GWT auch noch eine eigene UI Library, welche graphische Benutzerelemente enthält. Diese GUI Elemente sind meist schon in HTML (siehe http://www.gwtproject.org/doc/latest/DevGuideUiBrowser.html) verfügbar und werden als Java-Objekte zur Verfügung gestellt. Zudem werden viele Layouts angeboten um die Elemente platzieren zu können. Die GUI Elemente werden in späteren Kapiteln noch vertieft behandelt.

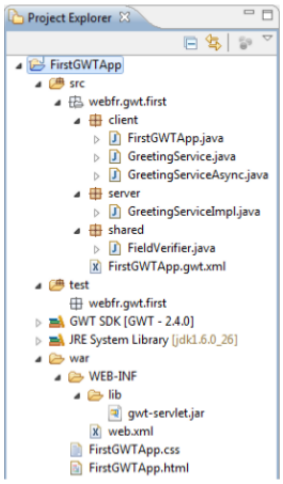


Abbildung 1.4.: figure

1.3.4. Projektstruktur

Wird ein GWT-Projekt mit einem Wizard angelegt (ob aus Eclipse oder direkt mit Hilfe des webAppCreator-Skriptes vom GWT-SDK), erhält es folgende Struktur:

src: Enthält die Sourcen und Modulkonfiguration der Applikation. Die Sourcen sind in drei Packages aufgeteilt:

- client: Code der im Browser ausgeführt und somit nach JavaScript compiliert werden muss.
- server: Code der auf dem Webserver läuft (Bytecode).
- shared: Code der auf beiden Plattformen laufen soll (Java Bytecode + JavaScript). Die Applikation kann innerhalb dieser drei Packages beliebig weiter verschachtelt werden. Code im shared und im client-Package unterliegt den oben erwähnten Beschränkungen des GWT Compilers.

Das schon erwähnte war-Verzeichnis enthält die „GWT Runtime“ in Form eines jar-Files. Der Web- Deskriptor (web.xml) beschreibt die Applikation für den Webserver (Details dazu haben Sie im 1. Teil des Moduls gehört). Zudem findet man hier auch die Host HTML Page und ein CSS File, dessen Styles aus dem Code referenziert werden können.

1.3.5. GWT Module

Die Basiseinheit in GWT ist ein Modul. Da Java kein Modulsystem hat, muss jedes Modul in einem .gwt.xml File beschrieben werden. In unserem Beispiel wäre dies FirstGWTApp.gwt.xml:

Listing 1.1: Modul Config GWT

```
1  <?xml version="1.0" encoding="UTF-8"?>
   <module rename-to='firstgwtapp'>
   <inherits name='com.google.gwt.user.User'/>
   <inherits name='com.google.gwt.user.theme.clean.Clean'/>
   <entry-point class='webfr.gwt.first.client.FirstGWTApp'/>
6  <source path='client'/>
   <source path='shared'/>
   </module>
```

Das optionale rename-to Attribut im module-Tag erlaubt es der Applikation einen „lesbaren“ Namen zu geben, denn sonst würde einfach der vollqualifizierte Packagename gelten: webfr.gwt.first.client.

Die ¡inherits¿ Elemente importieren den Inhalt anderer Module. Module können Applikationen oder Libraries sein. In diesem Fall wird eine Bibliothek mit Standard GWT- Funktionalität und eine mit GUI-Elementen importiert.

¡entry-point¿ Elemente definieren die Startklassen, deren onModuleLoad-Methode aufgerufen wird. Sind mehrere entry-points definiert, geschehen diese Aufrufe in der Reihenfolge der Deklaration im XML.

Die ¡source¿ Elemente geben an, welche Packages der GWT-Compiler nach JavaScript übersetzen muss.

1.3.6. Development Mode

- Im Production Mode wird die Applikation als JEE Webapplication auf einen Webcontainer (z.B. Tomcat) deployed. Die Applikation ist nach JavaScript übersetzt und dieser Code wird in den Browser geladen und dort ausgeführt.

- Im Development Mode wird kein JavaScript-Code erzeugt, sondern die Java .class Files werden in den Browser geladen und dort ausgeführt. Braucht plugin. Vorteile:

  Da der Code in Java ausgeführt wird, kann er auch mit normalen Java-Debuggern angehalten, durchgestept und inspiziert werden.

  Da der Code im Browser läuft, erhält man einen sehr genauen Eindruck davon, wie sich die Applikation später im Production Mode verhalten wird.

- Nach jedem neustart der App muss Plugin neugestartet werden. Verzögerung
- Im Dev mode ist es möglich nicht unterstutzte klassen zu verwenden.
- Serverseitig : Jetty im Dev Mode aber verhält sich anders wie Tomcat.
- Regelmässiges Deployment empfohlen.

2. Benutzer Interface

2.1. Nachteile von normallen GUI Programmieren

- Neukompilation notwendig bei Änderungen
- Programmierkenntnisse notwendig (OO)
- Struktur des GUIs nicht ersichtlich aus dem Code.
- Seperation of Concerns - keine Trennung

2.2. Vorteile deklarativen GUI

Eine graphische Benutzeroberfläche ist als Datenstruktur betractet, nichta anderes als ein Baum der aus GUI Elemente besteht. Bäume können auch in XML deklarativ beschrieben werden.

- Struktur GUI ersichtlich.
- Code zur Startzeit generiert. keine Kompilation bei änderungen.
- keine Programmierkenntnisse notwendig
- "Seperation of Concerns"
- Kompaktheit
- Schneller gerendert im Browser. Rendering effizientier als Manipulationen im DOM[1] tree.

2.3. Deklarative Layouts

Nebenan sieht man dass dieser Screen sich aus ein HTML Panel mit Textbox und Button zusammensetzt. Kommt aus XML File:

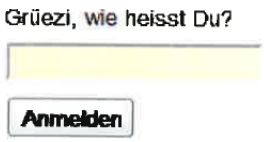


Abbildung 2.1.: figure

```
 <!DOCTYPE ui:UiBinder SYSTEM "http://dl.google.com/gwt/DTD/xhtml.ent">
<ui:UiBinder xmlns:ui="urn:ui:com.google.gwt.uibinder"
xmlns:g="urn:import:com.google.gwt.user.client.ui">
<ui:style>
.important {
font-weight: bold;
}
</ui:style>
<g:HTMLPanel>
<table>
<tr>
<td>Gr&uuml;ezi, wie heisst Du?</td>
</tr>
<tr>
<td><g:TextBox ui:field="textbox" styleName="hallo"/></td>
</tr>
<tr>
<td><g:Button styleName="{style.important}"
```

[1]Domain Object Model

```
ui:field="button" text="Anmelden" /></td>
</tr>
</table>
</g:HTMLPanel>
</ui:UiBinder>
```

Man spricht hier von einem Template, weil dieses XML-File als Vorlage für die HTML-Generierung dient. XML ist nicht HTML und kennt daher auch keine Sonderzeichen wie z.B. „ü“ bzw. ü. In Zeile 1 werden diese Entity-Definitionen importiert. Zeile 2 und 3 definieren XML-Namespace-Präfixe. Somit wird festgelegt, dass alle Java-Klassen aus dem Package com.google.gwt.user.client.ui als Elemente in XML verwendet werden können. Und zwar indem der Präfix mit dem Klassennamen kombiniert wird: g:TextBox oder g:Button. Analoges gilt auch für den Namespace ui. Zeilen 4-8 definieren einen CSS-Style namens important. Der Klassenselektor wird im folgenden Code verwendet, allerdings indem das Attribut styleName statt class gesetzt wird. Ab Zeile 9 wird der sichtbare Inhalt der Seite deklariert. Ein HTMLPanel demonstriert hier exemplarisch, wie HTML-Code verwendet werden kann und wie in diesem Code wiederum GWT-Tags vorkommen können. Das Loginformular besteht aus einer Tabelle, deren erste Zelle Text enthält. Dieser Text ist HTML-Text und wird daher (wie auch die Tabelle) nicht in der Strukturübersicht des GUI-Designers angezeigt. Zeile 15 deklariert eine TextBox, die einen Style namens „hallo“ verwendet. Dieser Style ist nicht lokal im selben XML deklariert. Er wird später aus dem Default-CSS-File importiert, welches im war-Verzeichnis liegt und den Namen des GWT-Modules trägt. Besonders interessant ist das Attribut ui:field: Es macht sein Element (hier die TextBox) über eine @UiField-Annotation in Java zugänglich. Dazu gleich mehr. In Zeilen 18 und 19 geschieht analoges mit einem Button. Hier sei angemerkt, dass die Beschriftung des Buttons auch so erfolgen könnte:

```
<g:Button styleName="{style.important}" ui:field="button">Anmelden</g:Button>
```

2.3.1. Nachteil : UI Binder

Eine GUI-Klasse besteht aus einem deklarativen Teil und aus einem imperativen Teil. Im deklarativen Teil wird das GUI deklariert. Diese Deklaration lässt sich schneller parsen und auswerten als der imperative Programmcode. **Es ist aber nur im Programmcode möglich Eventhandler zu installieren und somit das statische GUI zu beleben oder gar zu verändern.**

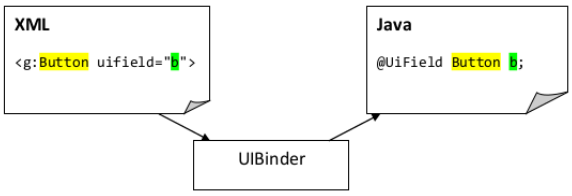


Abbildung 2.2.: figure

In GWT wird dieses Konzept mittels eines sog. UIBinders umgesetzt, der den XML-Code parsen und die darin befindlichen GWT-Elemente mittels Annotationen dem Java-Code zur Verfügung stellen kann. Für den imperativen Teil ist eine ganz normale Java-Klasse zuständig. Werden die beiden Teile, bis auf die Endung, gleich benannt (in unserem Fall LoginForm.ui.xml und LoginForm.java) erkennt sie der UIBinder automatisch.

```
 public class LoginForm extends Composite {
interface LoginFormUiBinder extends UiBinder<HTMLPanel, LoginForm> { }
private static LoginFormUiBinder uiBinder = GWT.create(LoginFormUiBinder.class);
public LoginForm() {
initWidget(uiBinder.createAndBindUi(this));
}
@UiField TextBox textbox;
public LoginForm(String firstName) {
initWidget(uiBinder.createAndBindUi(this));
textbox.setText(firstName);
}
@UiHandler("button")
void onClick(ClickEvent e) {
Window.alert("Hello!"); // process event
}
}
```

In Zeile 1 wird der Interface-Erweiterung von UiBinder¡U, O¿ definiert. U ist der Typ des Root- Objektes des generierten UI-Objektes (in unserem Fall ein HTMLPanel). O ist die „owning class“, also die Klasse, der dieses Binding

zugeordnet ist (LoginForm). In Zeile 3 wird dann ein Binding-Objekt mit Hilfe des zuvor definierten Interfaces erzeugt. Dieses Binding-Objekt wird später benötigt um das tatsächliche UI zu erzeugen (Zeilen 6 und 12). Die Methode createAndBindUi, der das „owning object“ übergeben wird, erstellt zugleich auch das sogenannte Binding. In Zeile 9 wird mittels der Annotation @UiField die deklarierte Variable an das XML-Element gebunden, dessen ui:field-Attribut gleich dem Variablennamen (case sensitive!) ist. Damit ist es z.B. in Zeile 13 möglich auf das TextBox-Objekt zuzugreifen, als ob das GUI mit Java-Code konstruiert worden wäre. Ähnliches geschieht in Zeile 16. Hier wird mit Hilfe der @UiHandler-Annotation die nachfolgende Methode an das GUI-Element gebunden, dessen ui:field-Attribut gleich dem angegebenen Namen ist. Hier wird allerdings die Bindung nicht auf eine Variable gemacht, sondern auf einen EventHandler. Es ist dabei wichtig, dass tatsächlich auch ein Handler mit dem Namen der nachfolgend definierten Methode existiert. Kurz: in den Zeilen 16 und 17 wird der Handler definiert und installiert, der bei einem Klick auf den Button ausgelöst wird.

Beachte: Es ist auf eine saubere Trennung der View vom Rest der Applikation zu achten. Darum sollte die Java- Klasse so wenig Code wie möglich enthalten. Sie repräsentiert das GUI und darf keinerlei Applikations- oder Controllerlogik enthalten. Hier sollen nur Initialisierungen und Registrierungen von Handlern vorgenommen werden. Handler sollten ihren Event direkt an eine Controllerlogik weiterleiten.

3. Model View Presenter Pattern

Klassiche MVC Pattern problematisch :

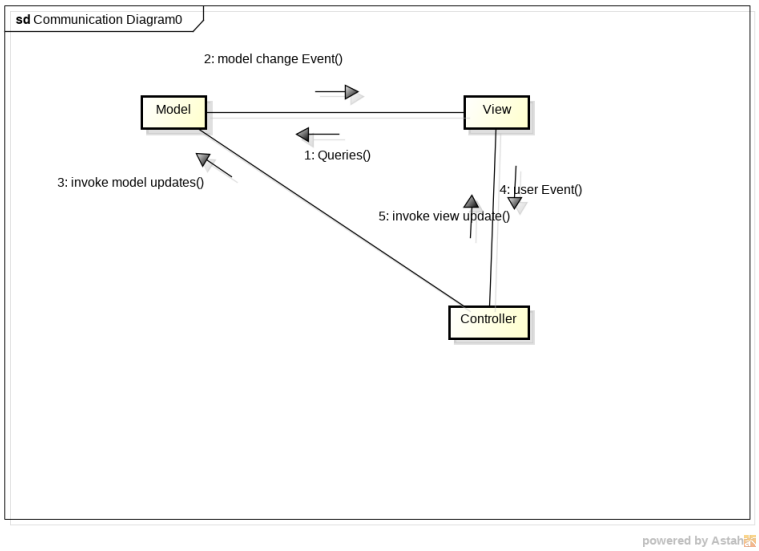


Abbildung 3.1.: figure

- Enge Koppelung der Komponenten
- Im Web : Model Veränderung können nicht am View übermittelt werden. (bei einem Request der View möglich)

Hilfe mit MVP:

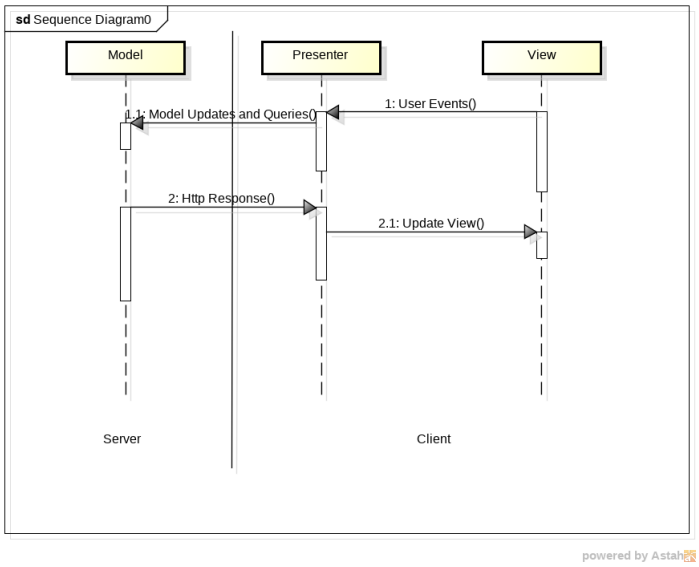


Abbildung 3.2.: figure

Model Hat dieselbe Rolle wie im MVC-Pattern, kommuniziert aber nur mit dem Presenter.

View: Hat auch eine ähnliche Rolle wie im MVC-Pattern, kommuniziert aber nicht mit dem Modell. Findet eine Benutzeraktion statt, meldet die View dieses Ereignis immer dem Presenter.

Presenter: Dient als einziges Bindeglied zwischen Modell und View. Der Presenter erhält Ereignisse von der View und leitet diese in Form eines Requests an das Modell weiter. Das Modell antwortet indem eine Callback-Methode aufgerufen wird (asynchron). In diesem Callback kann der Presenter die View veranlassen sich zu ändern. Dies geschieht indem er entsprechende Methoden auf der View aufruft und Daten vom Modell zur Darstellung weiterleitet.

3.1. Model

Ein Model setzt sich aus den Business-Objekten zusammen. Diese Objekte modellieren die Business- Logik und sind meistens auf dem Server angesiedelt. Selten findet man Modellklassen auf dem Browser. Im Falle der Lernkartei sind die Modellklassen (je nach getroffenem Entwurf): Bemerkung : Alles im Browser aber auch auf

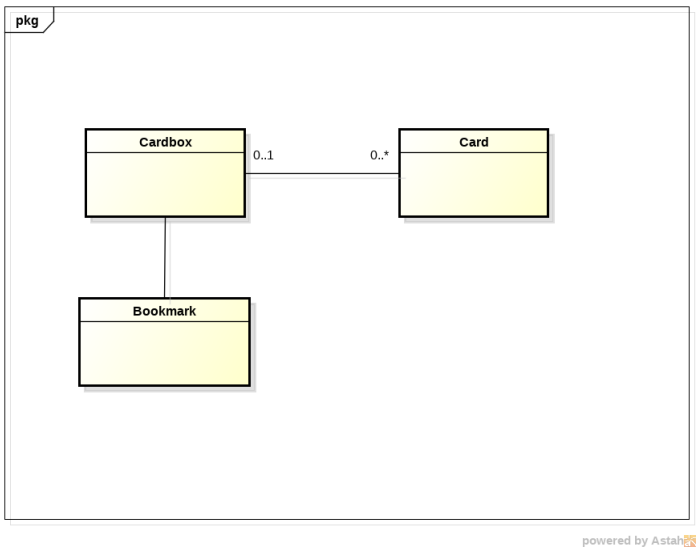


Abbildung 3.3.: figure

Server

3.2. View

Eine View enthält alle UI-Komponenten, die für die Bedienung eines bestimmten Teiles der Applikation benötigt werden. Views bestehen aus Tabellen, Buttons, Scrollbalken, Textfeldern, etc. und sind verantwortlich für das Layout der einzelnen UI-Komponenten. Sie haben keinerlei Vorstellung von einem Modell. Eine View weiss z.B. nicht, dass sie ein Kärtchen darstellt. Sie weiss nur, dass sie aus einem Label, einem Image, zwei Buttons, etc. besteht. Eine View besteht aus einer View-Klasse und einem zugehörigen ui.xml-File welches das Layout der View beschreibt. Über den UIBinder werden diese beiden Komponenten miteinander verbunden (siehe Kapitel 2). Views sollten keine Applikations- und schon gar keine Businesslogik enthalten. Allerdings müssen die Handler welche auf UI-Events reagieren in der View implementiert werden. Der Grund ist einfach: Der UIBinder ermöglicht nur in der gebundenen Klasse die Verwendung der @UIHandler Annotation. Wie soll nun die Implementation eines Handlers aussehen?

- Handler ruft Callback auf (z.B in Presenter)
- Callback mittels Setter Methode auf der View gestzt werden (Dependency Injection möglich)
- Browserseit hat ber kein DI

```
// im View
private ClickHandler okButtonClick;
public setOKButtonClickHandler(ClickHandler k) {
 okButtonclick = k;
}

 @UIHandler("okButton")
void onOKButtonClick(ClickEvent e) {
  if (okButtonClick != null) {
```

```
            okButtonClick.onClick(e);
        }
    }
```

3.3. Display(View Interface)

ews sind graphische Einheiten, die innerhalb einer HTML Seite ausgetauscht werden können. Um alle Views gleich behandeln zu können, sollten sie die asWidget()-Methode aus dem IsWidget Interface anbieten. Es drängt sich ein eigenes Interface Display auf, welches von jedem View- Presenter-Paar dann erweitert wird und die View-Details vom Presenter abstrahiert:

```
import com.google.gwt.user.client.ui.IsWidget;
public interface Display extends IsWidget { }
```

Damit steht jede View als Widget zur Verfügung. Ein Widget ist eine Basisklasse im GWT-UI- Framework, somit können alle UI-Elemente als Widgets interpretiert und manipuliert werden. Z.B. indem auf der HTML-Seite im DOM-Baum Widgets ausgetauscht werden, wenn ein Übergang zu einem anderen Teil der Applikation erfolgen soll. Jede konkrete View sollte gegen ein eigenes Display-Interface implementiert werden. Dieses spezifische Interface ermöglicht es dem Presenter z.B. Handler zu setzen, Texte anzupassen oder Farben zu setzen. Allgemein soll das Display nur gerade die Methoden deklarieren, die der Presenter benötigt um die View anzusteuern. Dies bedeutet, dass das spezifische Display-Interface dem Presenter auch als View-Abstraktion dient.

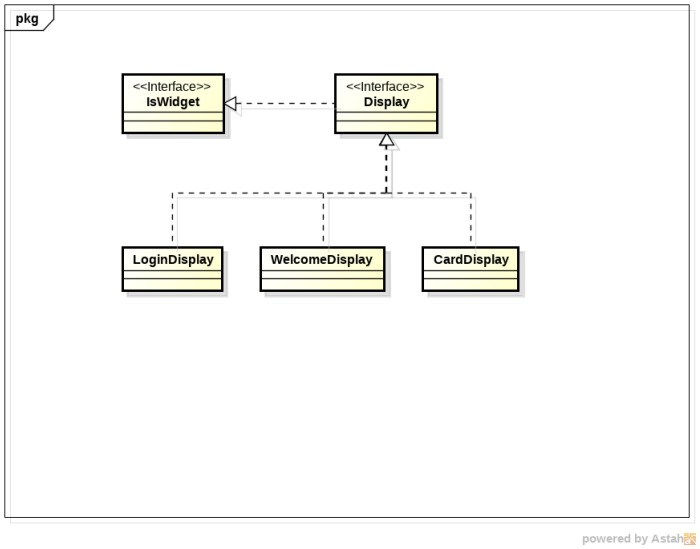


Abbildung 3.4.: figure

3.4. Presenter

Ein Presenter enthält die (Ablauf-)Logik der Applikation. Das beinhaltet auch das lokale History Management, Übergänge zwischen den Views und die Datensynchronisation mit dem Server. Als Faustregel gilt: Für jede View ist ein Presenter als „Treiber“ und „Event-Verarbeiter“ zuständig. Im Konstruktor eines Presenters kann die View als Argument übergeben werden, so dass die beiden von Anfang an miteinander verbunden sind. Ein Presenter sollte seine Views nicht direkt kennen, sondern nur über ein Display-Interface ansteuern. Dadurch werden UI-spezifische Klassen nicht mit Ablauf- oder Businesslogik vermischt. Ein späterer Austausch des GUI ist einfacher. Der Presenter kann auch mit dem Server kommunizieren und er ist für die Verarbeitung der Events aus der View verantwortlich. Beides wird später noch genauer behandelt.

Bind
- Bindet Handler + View an Presenter.
- Event Verdratung
- Display Services (RPC, Eventbus im Konstruktor gesetzt)

Present View wird in ein Container gesetzt.

GUI muss auch richtigen Inhalte presentieren/anziegen gemäss App Status z.B zeigt letzt benutzte Kartschen an.

Diese beiden Aufgaben können auch gleich als Basis dienen für ein Presenter-Interface. Dadurch können alle Presenter einer Applikation gemeinsam verwaltet und uniform angesteuert werden. Diese Eigenschaft ist vor allem für die History-Verwaltung und das Umschalten von einer View zur nächsten wichtig.

3.5. App Controller

Der AppController ist die Instanz, an der die ganze Applikation aufgehängt wird. Jemand muss die verschiedenen Views und Presenter erzeugen. Zudem wird der AppController das gesamte History Management der Applikation steuern sowie die globale Koordination der View-Übergänge übernehmen. Im AppController können auch Werte gespeichert werden, die für die ganze Applikation relevant sind. Z.B. ob ein erfolgreiches Login stattgefunden hat oder nicht. Die Rolle des AppControllers wird noch klarer, wenn im nächsten Kapitel das History-Management behandelt wird.

3.6. Event Bus

GWT bietet einen zentralen Event-Handling-Mechanismus an. Es handelt sich beim sogenannten Eventbus um eine Klasse, bei der sich interessierte Listeners registrieren können und über die dann Events an diese Listeners gesendet (abgefeuert) werden können. Das Methodenschnittstelle von com.google.web.bindery.event.shared.EventBus ist sehr einfach:

```
 // Adds an unfiltered handler to receive events of this type from all sources.
public <H> HandlerRegistration addHandler(Event.Type<H> type, H handler)
// Adds a handler to receive events of this type from the given source.
public <H> HandlerRegistration addHandlerToSource(Event.Type<H> type,
Object source, H handler)
// Invokes event.dispatch with handler.
protected static <H> void dispatchEvent(Event<H> event, H handler)
// Fires the event from no source.
public void fireEvent(Event<?> event)
// Fires the given event to the handlers listening to the event's type.
public void fireEventFromSource(Event<?> event, Object source)
// Sets source as the source of event.
protected static void setSourceOfEvent(Event<?> event, Object source)
```

Das HandlerRegistration-Objekt welches bei den addHandler-Methoden zurückgegeben wird erlaubt das spätere abmelden eines Listeners (removeHandle). Ein EventBus eignet sich nicht für alle Events. Oft ist es aus Gründen des Software Designs gar nicht wünschenswert lokale Events zentral über einen Hub zu verteilen. Events die über den Eventbus verschickt werden sollten von globalem Interesse sein. Beispiel: Wird eine Karte aus einem Karteikästchen gelöscht, so interessiert dieses Ereignis nicht nur die View über welche die Löschung veranlasst wurde, sondern auch die Views die z.B. die Frage-Antwort-Dialoge enthalten.

3.7. Applikationsüberblick

Das Gerüst steht! Jetzt müssen die einzelnen Komponenten nur noch richtig zusammengesetzt werden. Das Bootstrapping der Applikation könnte wie folgt geschehen:

1. In Flashcards.gwt.xml ist die Klasse Flashcards als EntryPoint vermerkt. Deren onModuleLoad() Methode wird aufgerufen.
2. onModuleLoad() erzeugt den Remote Procedure Call Service, den Event Bus, und den AppController.
3. Dem AppController wird die RootPanel Instanz übergeben und damit übernimmt er die Kontrolle über die Darstellung der Views.
4. Danach kontrolliert der AppController die Erstellung von spezifischen Presentern. Denen werden auch zugehörige Views zur Steuerung übergeben.
5. Eine von vornherein festgelegte Start-View wird vom AppController angzeigt.

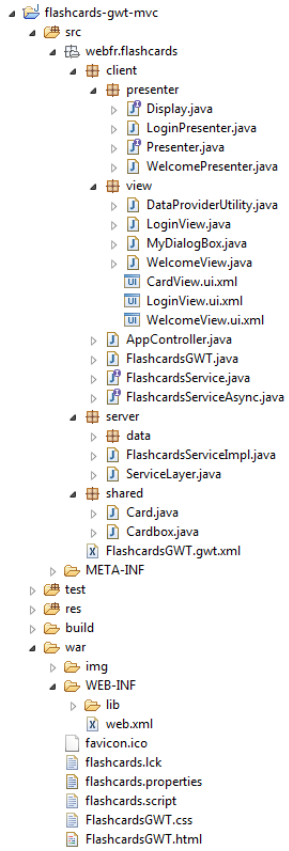
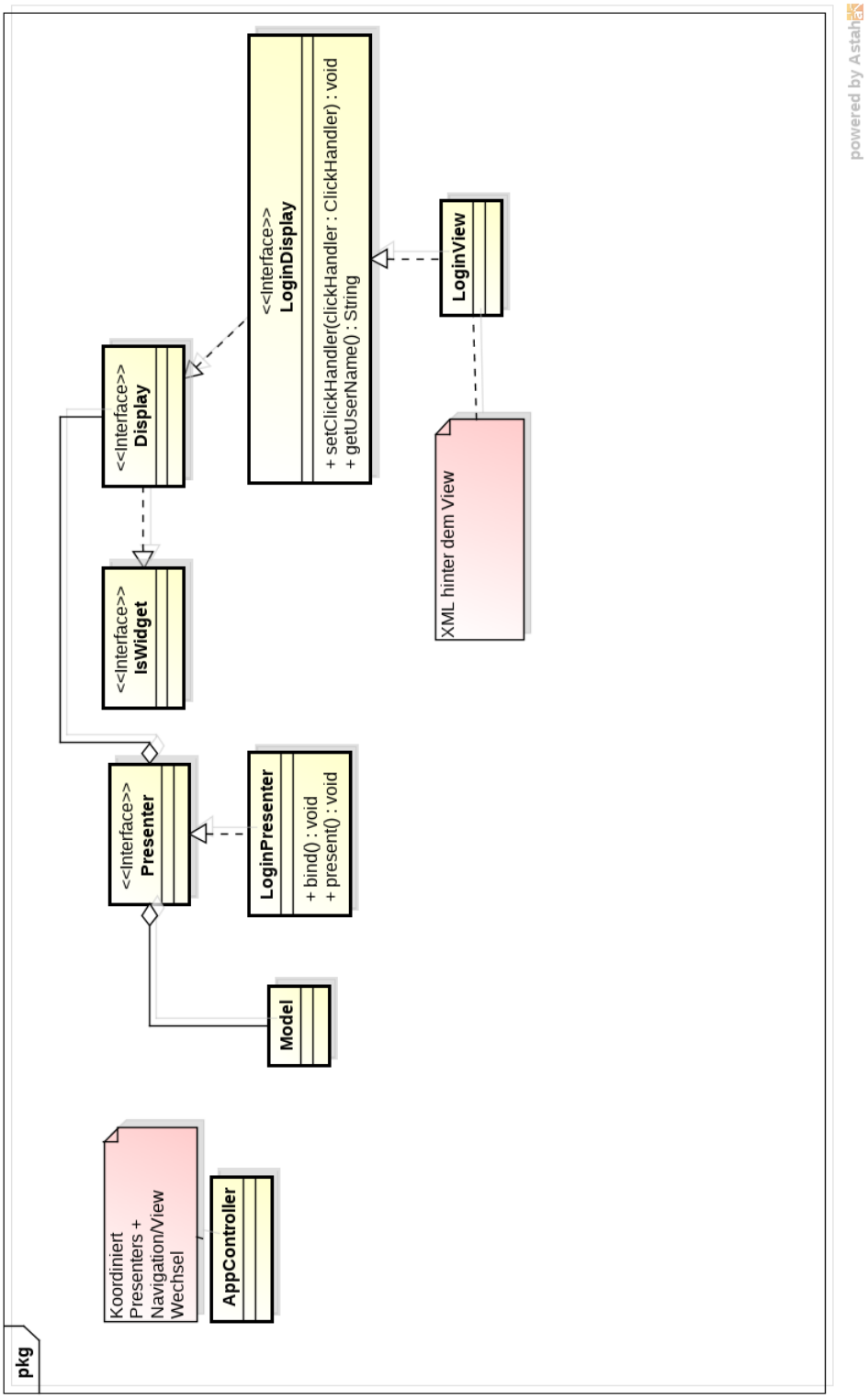


Abbildung 3.5.: figure

3.8. Wandtafel Extras

Eventbus:

- Eventhandler System, ort wo sich Events hin und her schicken - Observer Pattern - register Listener.
- Weniger Koppleung zwischen App Controller und Presenter durch kommunikation mittels Events.
- Events kann auch an mehere interessierte Parteien gemeldet werden.


pkg
<<Interface>>
LoginDisplay
+ setClickHandler(clickHandler : ClickHandler) : void
+ getUserName() : String
LoginView
<<Interface>>
Display
XML hinter dem View
<<Interface>>
IsWidget
<<Interface>>
Presenter
LoginPresenter
+ bind() : void
+ present() : void
Model
Koordiniert Presenters + Navigation/View Wechsel
AppController
powered by Astah

4. History Management

4.1. Back Button Problem

- Applikation mit GWT besteht nur aus eine Seite (Single Page - Hosting page)
- DOM Tree ständig verändert.
- Browser können nur ein Seitenwechsel zuverlässig detektieren. Browser können feststellen dass das DOM Tree sich verändert hat, aber können nicht der Grund der Veränderung feststellen (ob Navigation ist oder nicht z.B).
- Browser History verändert sich deshalb nur bei Seitenwechsel.
- Benutzer verlässt die Applikation dann ungewollt beim Klick auf dem Backbutton und verliert alle Zustandsinformationen. **Schwieriger History Management nachhinein einzubauen als von Anfgang an das dabei zu haben.**

4.2. History Klasse

GWT bietet dazu die Klasse com.google.gwt.user.client.History an, welche einfach aber effizient Events des Back- und Forward-Buttons verarbeiten kann.

4.2.1. URI Fragment als Token

Die Methoden der History-Klasse sind statisch und erlauben einen Handler für die Back- und For- ward-Buttons des Browsers zu registrieren. Um bei einer „Zeitreise“ mit Back oder Forward nicht gleich die HTML-Seite zu verlassen, verwendet ma
Auf dem History-Stack werden sogenannte Tokens gespeichert. Ein Token ist der Text der URI nach dem Hashzeichen. Es ist somit möglich jedem „zurückkehrbaren“ Zustand eine eindeutige URI zuzuordnen und sogar noch mit Parametern zu versehen. Dieser Ansatz hat folgende Vorteile:

- 1. Es ist die einzige zuverlässige Art die Browser-History zu kontrollieren
- Der Entwickler behält die Kontrolle darüber, welche Zustände in die History aufgenom- men werden.
- Es ist „bookmarkable“, d.h. ein Zustand lässt sich als Bookmark bzw. als Favorit speichern.

Bemerkungen:

- Die erwähnten Parameter können in beliebiger Form an das Token angehängt werden. Auf kei- nen Fall dürfen Sie vor dem Hashzeichen stehen.
- Hinter dem Token können aber beliebig viele Parameter z.B. in der Form `?key1=value1&key2=value2...` folgen. Es gibt keine Vorschrift zum Format der Parameter, aber auch keine Unterstützung bei deren Erkennung. D.h. Parameter müssen selbst geparst werden.

4.3. Value Change Handler

Wie kann ein Handler registriert werden, der aufgerufen wird, sobald sich die History ändert? Ein com.google.gwt.event.logical.shared
muss folgende Methode implementieren:

```
void onValueChange(ValueChangeEvent<T> event)
```

Wird nun die History geändert – sei es über die Back- und Forward-Buttons des Browsers, oder programmatisch – so erhält man im Handler einen ValueChangedEvent¡String¿. Mit des- sen getValue-Methode lässt sich das History-Token (inklusive eventueller Parameter) auslesen. Als Typ-Parameter kann String verwendet werden, so ist kein Typcast bei der Verwendung der getValue-Methode nötig:

static void	**back**() Programmatic equivalent to the user pressing the browser's 'back' button.
static void	**fireCurrentHistoryState**() Fire ValueChangeHandler.onValueChange(ValueChangeEvent) events with the current history state.
static void	**forward**() Programmatic equivalent to the user pressing the browser's 'forward' button.
static String	**getToken**() Gets the current history token.
static void	**newItem**(String historyToken) Adds a new browser history entry.
static void	**newItem**(String historyToken, boolean issueEvent) Adds a new browser history entry and fires ValueChaneEvent if requested.
static HandlerRegistration	**addValueChangeHandler**(ValueChangeHandler<String> handler) Adds a ValueChangeEvent handler to be informed of changes to the browser's history stack.

Abbildung 4.1.: Die History Klasse

```
 public void onValueChange(ValueChangeEvent<String> event) {
String token = event.getValue();
// ...

//Die Klasse AppController bietet sich als zentraler Ort f r die Verwaltung der
    Seiten berg nge an.
public class AppController implements Presenter, ValueChangeHandler<String> {
// ...
```

Warum sollte der AppController auch das Presenter Interface implementieren? Beim Aufstarten der Applikation übernimmt der AppController ähnliche Aufgaben wie ein Presenter: 1. Er muss alle MVP-Patterns instanziieren und „verdrahten“ 2. Er muss die Einstiegsseite anzeigen Wie könnte also eine Implementation dieser MVP-Patterns zusammen mit dem AppController aussehen?

4.4. App Controller Code

Konstruktor:

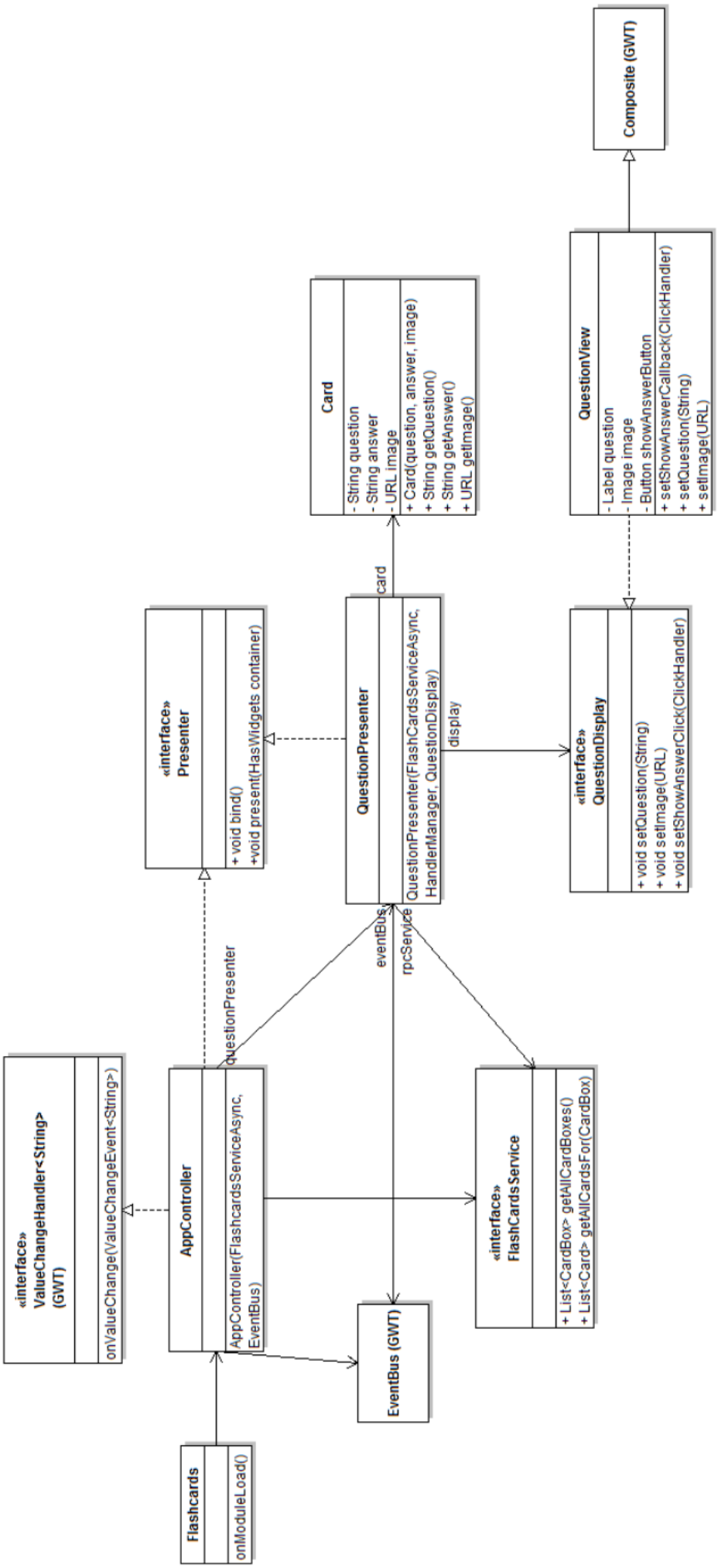
- Initalisiert Eventbus
- Services initialisieren (rpc)
- Bind();

  Presenter erzugen und Initialisieren.
- Registriert sich als ValueChangeListener bei History Klasse
- Registriert Handler im Eventbus.

```
onValueChange(Event<string> event){
 String token = event.GetValue();
 Presenter p = null;
 if (token == null || "".equals(token||"start".equals(token)){
   p = WelconePresenter(rpcService,eventBus,new WelcomeView());
   }else if ("question".equals(token)){
   p = QuestionPresenter(eventBus,new QuestionView();}else if..

   if (p != null) {
    p.present(container);
   }
 }
}
```

```
    //present methoden
    present(HasWidget container) {
    //immer noch im App Controller
18  this.container == container;
    if ("".equals(Histroy.getToken()){
     History.newItem("welcome"); // feuert ValueChangeEvent
    }else {
     History.fireCurrentHistoryState();
23  }
    }
```


Composite (GWT)
QuestionView
- Label question
- Image image
- Button showAnswerButton
+ setShowAnswerCallback(ClickHandler)
+ setQuestion(String)
+ setImage(URL)
Card
- String question
- String answer
- URL image
+ Card(question, answer, image)
+ String getQuestion()
+ String getAnswer()
+ URL getImage()
card
«interface»
Presenter
+ void bind()
+void present(HasWidgets container)
QuestionPresenter
QuestionPresenter(FlashCardsServiceAsync, HandlerManager, QuestionDisplay)
display
«interface»
QuestionDisplay
+ void setQuestion(String)
+ void setImage(URL)
+ void setShowAnswerClick(ClickHandler)
eventBus
rpcService
questionPresenter
«interface»
ValueChangeHandler<String>
(GWT)
onValueChange(ValueChangeEvent<String>)
AppController
AppController(FlashcardsServiceAsync, EventBus)
«interface»
FlashCardsService
+ List<CardBox> getAllCardBoxes()
+ List<Card> getAllCardsFor(CardBox)
EventBus (GWT)
Flashcards
onModuleLoad()

5. Kommunikation mit Server

5.1. Einführing - Erklärung von AJAX Request

Die meisten JavaScript Engines in den Browsern sind single-threaded, d.h. sie können nicht mehrere Aufgaben gleichzeitig (bzw. quasi-gleichzeitig) erledigen. Allerdings gibt es in JavaScript die Möglich- keit http-Requests asynchron abzusetzen. Das bedeutet, dass der Request an den Server geschickt wird und der Kontrollfluss unmittelbar danach wieder zum Aufrufer zurückkehrt, ohne dass eine Antwort (Response) vom Server abgewartet wird. Kommt nun die http-Response vom Server zurück, muss das Ergebnis irgendwie verarbeitet werden. Da das Ergebnis nicht wie ein Return-Wert zurück- gegeben werden kann (der Kontrollfluss wartet das Ergebnis nicht ab) bleibt nur die Variante einen zuvor installierten Callback-Handler aufzurufen.

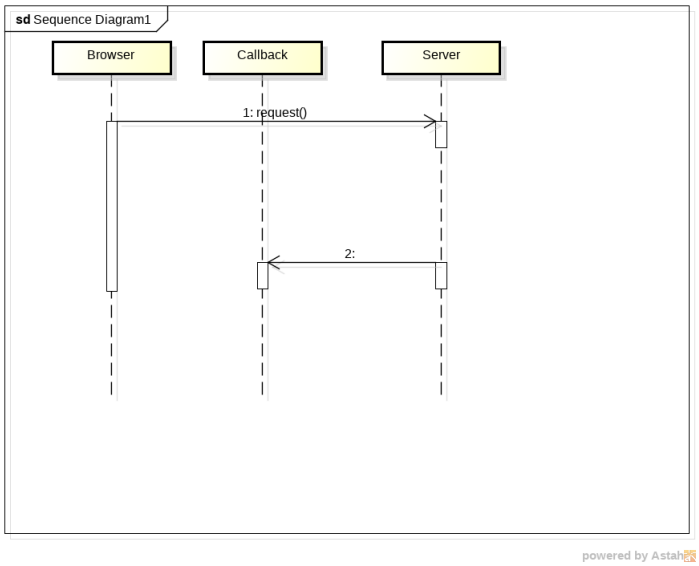


Abbildung 5.1.: figure

5.1.1. Vorteile AJAX

- Ohne Ajax wird die Javascript Engine blokieren bis Resultät kommt. **Javascript ist singleThreaded**
- Server kann unnereichbar sein oder aus anderen Grunden nicht Antwoerten - Wie soll Client damit umgehen ohne ajax?
- Ajax wird parallel zu Client Logik abgearbeitet, und Wartezeiten weden verkürtzt. (Beispiel : Rendering während wir auf Daten vom Server warten.)

5.2. RPC - Remote Procedure Calls

Für die Kommunikation mit einem Server bietet GWT – in Form von sogenannten Remote Procedure Calls – eine sehr komfortable Unterstützung. Trotz des Namens unterscheiden sich RPC in GWT von RPC in Unix oder RMI (Remote Method Invocation) in Java. Der Hauptunterschied ist, dass Remote Procedure Calls in GWT immer asynchron ausgeführt werden. Zudem sind RPCs immer nur in eine Richtung möglich, nämlich vom Client zum Server. Diese Ein- schränkung ergibt sich durch die Verwendung des http-Protokolls für sämtliche Kommunikation zwi- schen dem Browser und dem Server.

5.2.1. RPC Pattern

Serverseitig spricht man üblicherweise von Services, die gegenüber den Clients angeboten werden. Ein Service ist ein Java-Interface, welches mehrere thematisch zusammengehörende Methoden ent- hält. Wie werden Services deklariert? Folgende Grafik soll eine Übersicht geben: Jeder Service wird durch eine kleine Gruppe an Hilfsinterfaces und -

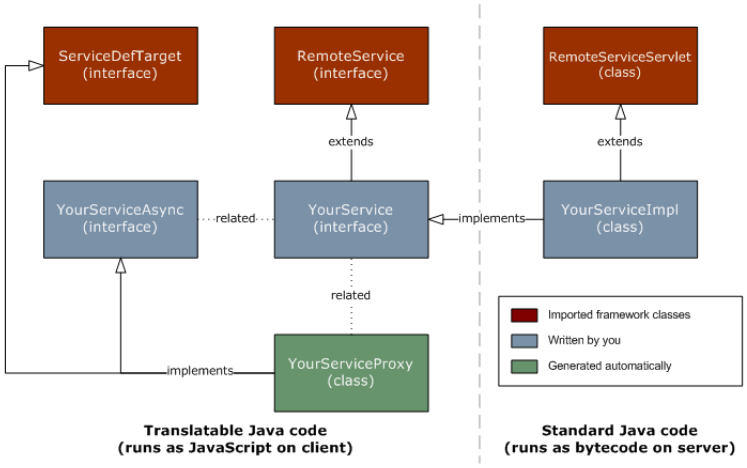


Abbildung 5.2.: figure

klassen implementiert. Auf der Client-Seite müssen nur zwei Interfaces zur Verfügung gestellt werden damit ein Stub generiert wer- den kann (in Form einer Proxy-Klasse). Der Stub ist für die Umwandlung des Methodenaufrufes in einen Http-Request zuständig. Zudem wird im Stub der Request asynchron abgesetzt und auf eine Server-Response gewartet. Diese Antwort wird dann wieder deserialisiert und in einen Methoden- aufruf des zuvor registrierten Callbacks umgesetzt.

5.2.2. RPC mit GWT

Um ein Service-Interface zu definieren sind folgende Schritte nötig:

1. Definiere ein Java-Interface, das com.google.gwt.user.client.rpc.RemoteService erweitert und alle benötig- ten Service-Methoden enthält.
2. Implementiere eine Klasse, die den Server-seitigen Code enthält, von com.google.gwt.user.server.rpc.RemoteServiceServlet abgeleitet ist und das Interface von Punkt 1 implementiert.
3. Definiere ein asynchrones Interface zum Service, welches client-seitig aufgerufen werden kann.

5.3. Namenskonventionen GWT RPC

Um dem GWT-Compiler die korrekte Generierung von RPC-Code zu ermöglichen, ist es notwendig einige Namenskonventionen einzuhalten. Dies betrifft vor allem die beiden Interfaces auf Clientseite:

- Ein Serviceinterface muss ein entsprechendes asynchrones Interface mit demselben Namen und der Endung „Async“ haben. Zudem ist es notwendig, dass beide Interfaces im gleichen Package deklariert werden. Beispiel: Ein Serviceinterface ch.fhnw.webfr.client.FlashcardService muss ein asynchrones Interface ch.fhnw.webfr.client.FlashcardServic haben.
- Jede Methode im synchronen Interface muss eine entsprechende Methode im asynchronen Interface mit einem zusätzlichen letzten Argument vom Typ com.google.gwt.user.client.rpc.AsyncCallback¡T¿ haben.

5.4. Synchrone Interfaces mit GWT

Das synchrone Interface spezifiziert den Service! Das asynchrone Interface muss sich danach richten. Eine Implementation des Services auf Serverseite muss das synchrone Interface implementieren. Wichtig: dieses Interface kann nicht direkt aufgerufen werden! Ein Aufruf muss über ein asynchrones Interface erfolgen.

```
1  package ch.fhnw.webfr.flashcards.client;
   import java.util.List;
   import com.google.gwt.user.client.rpc.RemoteService;
   @RemoteServiceRelativePath("fcService")
   public interface FlashcardsService extends RemoteService {
6  List<CardBoxInfo> getAllCardboxes(int minNofCards) throws IllegalArgumentException;
   }
```

5.5. Asynchrone Interface mit GWT

Ein asynchrones Interface ist nötig, da alle RPCs asynchron abgesetzt werden. Ein asynchroner Aufruf wird immer sofort zum Aufrufer zurückkehren und somit machen Return-Werte auch keinen Sinn. Die Methoden in diesem Interface sind daher immer void. Um Ergebnisse aus dem Aufruf trotzdem verarbeiten zu können, gibt es den Callback-Handler. Das entsprechende asynchrone Interface muss demnach so aussehen:

```
   package ch.fhnw.webfr.flashcards.client;
   import java.util.List;
3  import com.google.gwt.user.client.rpc.AsyncCallback;
   public interface FlashcardsServiceAsync {
   void getAllCardboxes(int minNofCards, AsyncCallback<List<CardBoxInfo>> callback);
   }
```

5.6. Callback Handlers

Der Callback-Handler wird im asynchronen Interface als letztes Argument übergeben. Er steht anstel- le eines Rückgabewertes und hat auch dessen Typ im Typparameter. Ein Callback-Handler wird auf der Clientseite aufgerufen, sobald die Response vom Server beim Client ankommt. Ein Methodenauf- ruf kann nur zwei Ergebnisse haben: Ein erwartetes Resultat oder einen Fehler (ein abnormes Ereig- nis ist aufgetreten). Diese zwei Ergebnistypen werden durch die beiden Methoden des Typs AsyncCallback¡T¿ repräsentiert.

```
   package com.google.gwt.user.client.rpc;
   public interface AsyncCallback<T> {
   void onFailure(java.lang.Throwable caught);
4  void onSuccess(T result);
   }
```

Der Typparameter ¡T¿ entspricht dem Return-Typ der Methode im synchronen Interface. Die onSuccess-Methode erhält als einziges Argument das Resultat des Methodenaufrufes. Sie wird im Falle einer normalen Ausführung der Methode aufgerufen. Die onFailure-Methode wird immer dann aufgerufen, wenn in der Methode selbst eine Exception geworfen wurde oder wenn es Probleme mit dem Aufruf oder der Antwort bzw. deren Übermittlung gab (z.B. Timeout).

5.7. Implementation der Interface

Die Implementation der Servicemethoden erfolgt in einer Server-Klasse, die gleichzeitig auch ein Servlet ist. Dazu wird die Klasse RemoteServiceServlet erweitert und das synchrone Interface implementiert:

```
   package ch.fhnw.webfr.flashcards.server;
   // import statements here
   public class FlashcardsServiceImpl extends RemoteServiceServlet
   implements FlashcardsService {
5  @Override
   public List<CardBoxInfo> getAllCardboxes(int minNofCards) throws IllegalArgumentException
   {
   return new LinkedList<CardBoxInfo>();
   }
10 }
```

Die Basisklasse RemoteServiceServlet kümmert sich um die Deserialisierung der Argumente (http-Request) und die Serialisierung (http-Response) der Return-Werte. In den meisten Fällen hat der Entwickler nichts mit diesen oder anderen Servletmethoden zu tun.

5.8. Parameter und Ruckgabetypen

Sämtliche Argumente sowie die Rückgabewerte müssen über das Netzwerk vom Browser zum Server und zurück. Dies bedeutet, dass alle Argumente und Rückgabewerte serialisierbar sein müssen. Auf folgende Punkte ist zu achten:

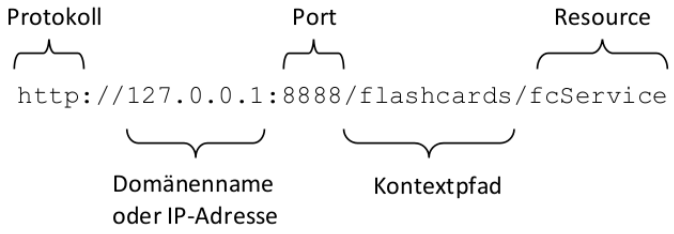
- Alle Klassen die als Parameter oder Rückgabetypen verwendet werden, müssen java.io.Serializable implementieren.
- Alle Klassen die als Parameter oder Rückgabetypen verwendet werden, müssen einen Default-Konstruktor (d.h. einen Konstruktor ohne Argumente) oder gar keinen Konstruktor haben.
- Falls Collections verwendet werden, sollte unbedingt der Typ der Elemente angegeben werden, also z.B. List¡CardBoxInfo¿. Die Verwendung von rohen Typen oder von Object funktioniert nicht, da java.lang.Object selbst nicht Serializable ist.

5.9. RPC Aufruf Prozess von Client aus

```
 public void makeCall() {
// (1) create a stub object that implements the asynchronous interface.
FlashcardsServiceAsync service = GWT.create(FlashcardsService.class);
// (2) Define a callback handler.
AsyncCallback<List<CardBoxInfo>> callback = new AsyncCallback<List<CardBoxInfo>>() {
@Override
public void onFailure(Throwable caught) {
// do some stuff on failure.
}
@Override
public void onSuccess(List<CardBoxInfo> result) {
// use result for further processing.
}
};
// (3) make the call.
service.getAllCardboxes(0, callback);
// remember this call was asynchronous! Control flow will continue
// immediately here.
nextStatement();
}
```

1. Zuerst muss der Service mittels GWT.create()instanziiert werden.
2. Dann muss ein asynchrones Callback-Objekt erzeugt werden, welches benachrichtigt wird, wenn der RPC beendet ist.
3. Den Aufruf über das asynchrone Interface machen.

5.10. Serveradresse für Aufrufe



ɜ Resource wird mittels einer Annotation vermerkt (siehe auch Abschnitt 2.2):

```
@RemoteServiceRelativePath("fcService")
```

Abbildung 5.3.: figure

Listing 5.1: RPC App Deployment Descriptor

```
 <servlet>
<servlet-name>cardsServlet</servlet-name>
<servlet-class>
ch.fhnw.webfr.flashcards.server.FlashcardsServiceImpl
```

```
</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>cardsServlet</servlet-name>
<url-pattern>/flashcards/fcService</url-pattern>
</servlet-mapping>
```

Der Kontextpfad /flashcards im URL-Pattern muss mit dem Namen der Applikation im Konfigu- rationsfile Flashcards.gwt.xml in der Moduldeklaration vermerkt sein

6. Deployment

Das Laden einer Webapplikation geschieht immer wenn ein/e Benutzer/-in wartet. Zudem wird die Applikation über das Internet und nicht von lokalen Datenträgern geladen. Um lange Wartezeiten zu vermeiden versucht man bei grösseren Applikationen diese in mehreren Schritten zu laden und zwar nur dann wenn die einzelnen Teile auch wirklich gebraucht werden.

6.1. Defered Binding

Der GWT-Compiler erzeugt immer ein .war-File, das neben den Javascript-Files auch die Java-Class- Files (für den Server) sowie auch alle nötigen Ressourcen (Bilder, Icons, Audio- und Video, etc.) ent- hält. Der Compilationsvorgang in GWT ist nicht nur deswegen aufwändiger und langsamer als der von normalen Java-Applikationen. Denn zusätzlich werden mehrere Compilate erstellt für verschiedene Browsertypen und verschiedene Locales.

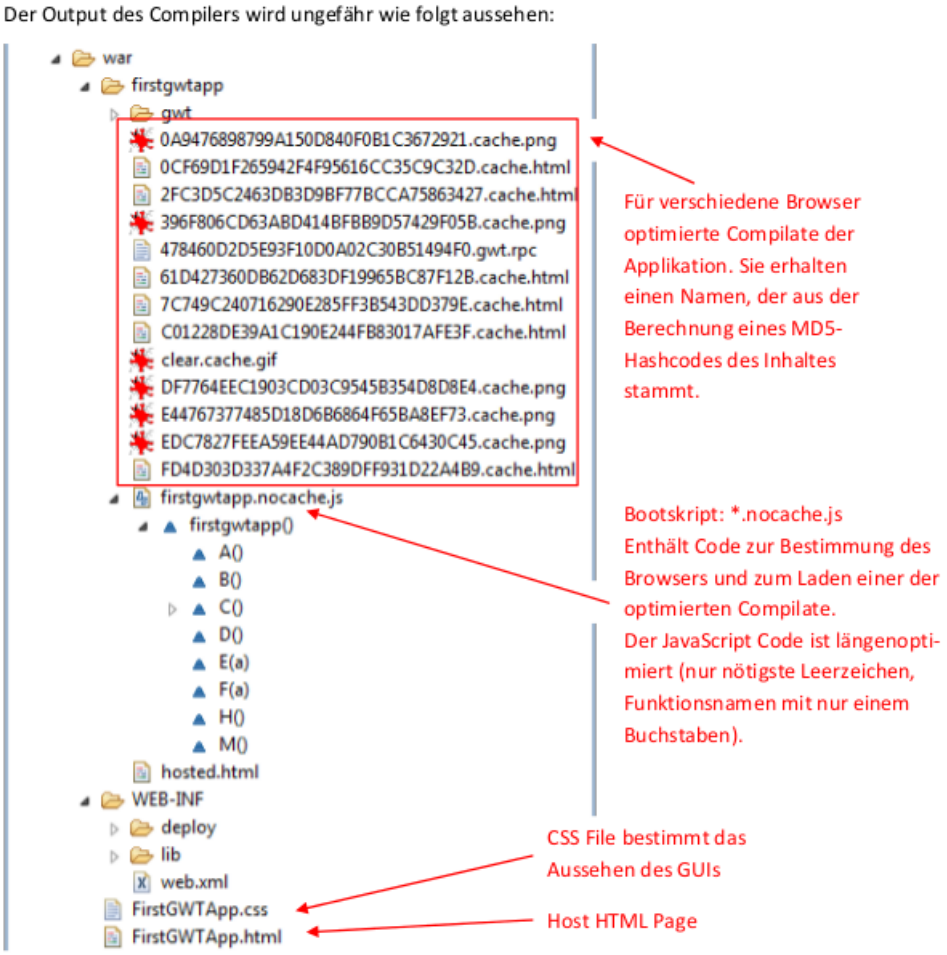


Abbildung 1: GWT Applikation auf dem Server

Listing 6.1: GWT Einstiegsseite

```
 html>
<head>
<meta http-equiv="content-type" content="text/html; charset=UTF-8">
<link type="text/css" rel="stylesheet" href=" FirstGWTApp.css">
<title>First GWT Application</title>
<script type="text/javascript" src='firstgwtapp/firstgwtapp.nocache.js'/>
</head>
```

```
  <body>
  <noscript><div>Please enable JavaScript</div></noscript>
10 <!-- more body content here.
  -->
  </body>
  </html>
```

Das angegebene Skriptfile wird geladen und ausgeführt. In diesem Bootskript wird der aktuelle Browser und dessen Version, sowie das eingestellte Locale bestimmt. Danach wird das dazu passen- de ¡md5¿.cache.html nachgeladen. Dieses ist eines der optimierten Compilate. Eine detaillierte Beschreibung dieses Bootstrap-Prozesses findet sich hier: `http://code.google.com/webtoolkit/doc/latest/DevGuideOrganizingProjects.html#DevGuideBootstrap`

Die Namensgebung ist nicht zufällig und erlaubt das Caching der Seiten zu optimieren. Bei jeder Compilation werden neue MD5-Hashcodes berechnet und die Applikationsfiles danach benannt. Wichtig ist nun, dass das *.nocache.js-File ein sofortiges Verfallsdatum erhält. D.h. der Webser- ver sollte diesen Files eine Gültigkeitsdauer von 0 Sekunden geben (¡meta http- equiv=ëxpirescontent=”0≫ im HTML-HEAD-Tag), damit die Browser das Bootskript nicht im Cache behalten. Bei jedem Anwählen der Applikation wird nun das *.nocache.js Skript neu geladen. Falls sich nichts geändert hat, wird im Browser die lokal gecachte Applikationsdatei ausge- führt (die verfällt z.B. innerhalb eines Jahres). Bei einer Änderung wird aber das Applikationsfile ei- nen neuen MD5-Code und damit einen neuen Namen erhalten. Der Browser wird es somit nicht im Cache finden und automatisch die aktuelle Version herunterladen und ausführen. Dies ist vor allem beim Debugging sehr praktisch, da keine Caches geleert oder Browser neu gestartet werden müssen. Ein Reload der Seite genügt. Dieser Vorgang des Ladens einer Applikation nennt sich Deferred Binding.

6.2. Compilation

Der GWT Compiler erstellt pro Browserfamilie und Locale jeweils ein Compilat. Es enthält den gesam- ten Applikationscode ohne unnötige Leerzeichen und mit möglichst kurzen Bezeichnernamen. Zudem enthält es auch alle für das eingestellte Locale relevanten Texte und Medienfiles. Damit werden gleich drei Anforderungen auf einmal erfüllt:

1. Die Ladezeit wird verkürzt weil die Files kleiner sind.
2. Der Code wird kleiner, weil keine unnötigen Übersetzungen und (übersetzte) Medienfiles übertragen werden müssen.
3. Der Code läuft schneller, weil Fallunterscheidungen entfallen.

Die Abbildung 2 zeigt in welcher Reihenfolge diese Compilate erstellt werden. Während der Entwicklung sollte immer wieder mal eine Compilation durchgeführt werden um mög- lichst früh herauszufinden, ob keine zur JRE Emulation inkompatiblen Sprachkonstrukte oder Biblio- theksmodule verwendet wurden.

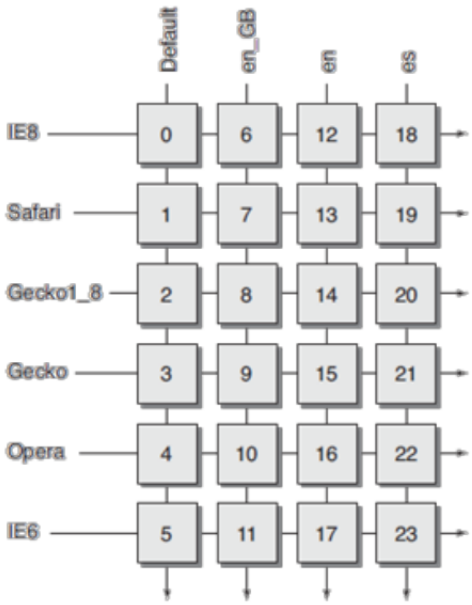


Abbildung 2: Compilate für die sechs Browsertypen und vier Locales. Die Compilate werden wie angegeben nummeriert, die tatsächliche Anzahl hängt allerdings von Compileroptionen und Sprachunterstützung ab.

Abbildung 6.1.: Compilate Matrix

```
   Falls der Code so schnell wie m glich compiliert werden soll, dann sollten folgende
       Optionen in Be-
2  tracht gezogen werden um Optimierunen auszuschalten:
   -draftCompile   optimize   0
   Zudem sollte im .gwt.xml File noch folgende Zeilen eingef gt werden damit nur eine
       Codeversion
   produziert wird:
   <set-property name="user.agent" value="gecko"/>
7  <set-property name="locale" value="en" />
   Andererseits bieten die Optionen
   -XdisableCastChecking    XdisableClassMetadata    -style OBF
   eine gute Ausgangslage f r m glichst kompakten Code, so wie man ihn f r die Release-
       Version ha-
   ben will. Die wenigen dokumentierten Compileroptionen sind unter folgendem Link zu finden:
12 http://code.google.com/webtoolkit/doc/latest/DevGuideCompilingAndDebugging.html#
       DevGuideCompilerOptions.
```

6.3. Code Splitting

Wenn eine Applikation grösser wird (man beachte den Gebrauch von wenn anstelle von falls :-) kann der initiale Download zu spürbaren Wartezeiten führen. Um dies zu verhindern kann eine Applikation beim Starten vorerst nur einen Teil laden. Danach werden dann bei Bedarf oder im Hintergrund all- mählich weitere Teile nachgeladen. Was ist der Unterschied zum Deferred Binding?

- Der Applikationsprogrammierer /-designer legt explizit fest, wo das Code Splitting zum Einsatz kommt, das Deferred Binding ist eine GWT Optimierung die automatisch erzeugt wird.
- Code Splitting teilt die gesamte Applikation in mehrere Teile auf während Deferred Binding die Aufteilung gemäss Browserfamilien vornimmt.

Code Splitting ist eine oft verwendete Technik auch in herkömmlichen Ajax Applikationen um die Wartezeit beim Aufstarten einer Applikation zu verringern. Code Splitting kann auf zwei verschiedene Arten eingesetzt werden. Dabei wird immer zuerst ein erster Teil der Applikation geladen, z.B. eine Welcome-View die dem Benutzer mehrere Optionen bietet. Danach kann wie folgt fortgefahren werden:

- Während der User noch über seine Möglichkeiten nachdenkt oder erste Eingaben auf der Wel- come-View macht, wird im Hintergrund (asynchron!) der Rest der Applikation geladen. Mit diesem Ansatz wird die Zeit bis die komplette Applikation im Browser ist zwar objektiv ver- längert (das Laden von mehreren Code-Paketen ist immer langsamer als das Laden eines einzigen Paketes). Subjektiv nimmt der Benutzer diese Wartezeit aber nicht wahr, da er z.B. mit der Ein- gabe seines Passwortes beschäftigt ist.
- Die weiteren Teile der Applikation werden erst bei Bedarf geladen, also erst dann wenn der Be- nutzer z.B. darauf klickt. Im Gegensatz zum ersten Ansatz wird hier Speicher und Bandbreite ge- spart, da nur diese Teile der Applikation heruntergeladen werden, die auch tatsächlich benötigt werden. Allerdings kann es so immer wieder zu Wartezeiten kommen. Beispiel zur Lernkartei (Flashcards): wenn nur gelernt wird, braucht es kein Code zum Editieren von Karten.

Die Kosten für einen Codesplit müssen nur einmal bezahlt werden: Wenn ein Teil der Applikation einmal heruntergeladen wurde, dann wird er nicht ein weiteres Mal beim Server angefordert.

7. Testing

7.1. Herkömmliche Ajax Apps

Testen von herkömmlichen RIAs ist sehr aufwändig da deren client-seitiger Code in Javascript entwi- ckelt wurde. Um diesen Javascript-Code zu testen muss eine Javascript Engine zur Verfügung stehen. Alle Tests müssen zuerst auf die JS-Engine deployed werden, bevor sie ausgeführt werden können. Daraus ergeben sich folgende Probleme:

- Lässt sich ein Test oder eine Testsuite automatisiert (z.B. mit Ant) deployen?
- Ist die JS-Engine in der Lage verschiedene Browser zu simulieren?
- Ist es möglich die verschiedenen Browser z.B. mittels Plugins zu erweitern, so dass ein auto- matisiertes Deployment durchführbar ist. Sind die Schnittstellen für solche Plugins auf den unterschiedlichen Browsern wenigstens einheitlich ansprechbar?
- Stehen vernünftige Debugger für die JS-Engine zur Verfügung? D.h. kann der Code schrittwei- se durchlaufen und können Variableninhalte inspiziert werden?
- Werden .js-Files unterstützt? Oder müssen Tests in html-Files verpackt werden?
- Kann dasselbe Testing-Framework sowohl Server- als auch Client-seitig verwendet werden?
- Stehen Testing Tools wie Mock-Frameworks, Code-Coverage oder Metrik-Tools zur Verfü- gung? Auf allen wichtigen Browsern? Mit gleichen Schnittstellen, damit automatisiert getes- tet werden kann?

Leider müssen oft gleich mehrere dieser Fragen mit „Nein“ beantwortet werden. Das Testen von JS- Code wird so zu einer sehr mühsamen Angelegenheit, die aufwändig zu implementieren und durch- zuführen ist. Als Konsequenz wird dann häufig auf Tests verzichtet, gerade mit dem Hinweis auf die hohen zusätzlichen Aufwände.

7.2. GWT Testing

Ganz anders sieht es für GWT-Applikationen aus: da der grösste Teil der Entwicklung in Java erfolgt, können zu grossen Teilen auch die gewohnten Java Testing Tools zum Einsatz kommen:

- JUnit als Basisframework – es wird nicht nur für Unit-Tests eingesetzt, sondern dient häufig auch als Treiber- plattform für Integrations- oder GUI-Tests. EasyMock, Mockito, jMock, etc.
- Code Coverage mit Emma, Cobertura, etc.
- Bug-Finder und Stylechecker wie Checkstyle, Findbugs, PMD
- Und viele andere Tools, die häufig dann auch noch aus der gewohnten Entwicklungsumge- bung wie Eclipse, Netbeans oder IntelliJ verwendet werden können (im Gegensatz zu Brow- serplugins)

Doch es ist nicht alles mit Java und den gewohnten Tools machbar. Einige Einschränkungen existie- ren:

- Tests von Klassen oder Komponenten, die GWT-Libraries verwenden (diese stehen in J2SE- Umgebungen nicht zur Verfügung), insbesondere GUI-Tests!
- Test der Kommunikation zwischen Client und Server
- Tests von Servlets (serverseitiger Code) die einen Servlet-Container (z.B. Tomcat) benötigen.

7.3. Vorbereitung Arch und Design

Eine gute Planung der Architektur einer GWT-Applikation hilft enorm dabei, möglichst viel Code mit Unittests zu versehen. Zwei Entwurfsmuster sind dabei besonders hervorzuheben:

1. Das MVP-Pattern hilft Verantwortlichkeiten aufzutrennen. Das Modell kann meist vollständig in Plain Java entwickelt werden, ohne zusätzliche Unterstützung durch GWT-Libraries oder Servlet-Container. Der Presenter kann zu einem grossen Teil auch komplett in Plain Java entwickelt werden. Dazu ist es nötig eine klare Trennung von der View mittels Display- Interfaces vorzunehmen. Dadurch kann die View mit Mock-Objekten „simuliert“ werden und es sind keine GWT-Libraries notwendig.

2. Dependency-Injection unterstützt das oben erwähnte Mocking. Als Beispiel: ein Presenter er- zeugt nicht selbst eine View, sondern erhält diese mittels einem Display-Argument über ei- nen Setter oder direkt in den Konstruktor.

Weitere Entwurfsmuster wie Factories, State-/Strategy-Pattern oder Decorators helfen Objekte un- abhängig von ihrem konkreten Typ zu erstellen, zu verwalten oder einzusetzen.

Wenn die Architektur einer GWT-Applikation zum vornherein die gängigen Entwurfsprinzipien ein- hält ist also ein sehr grosser Teil der Applikation mit herkömmlichen Mitteln testbar. Wie steht es aber mit dem Rest der Applikation, der nicht ohne GWT-Libraries oder Servlet-Container auskommt?

7.4. GWTTestCase

GWT verwendet HTMLUnit als eingebauter Browser und da HTMLUnit in Java geschrieben ist, ist es möglich Tests im Development-Mode zu debuggen. In Integrationstests ist es sogar möglich Klassen die Javascript verwenden zu testen. Allerdings gibt es auch Nachteile: Da es zwischen dem Development-Mode und dem Production- Mode subtile Unterschiede gibt (z.B. die unvollständige JRE-Emulation), können nicht alle Probleme mit GWTTestCase erkannt werden. Zudem ist es nicht möglich Mocking-Frameworks wie EasyMock einzusetzen (weil diese Reflection verwenden, was in der JRE-Emulation nicht unterstützt wird).

7.5. Selenium

Selenium ist ein Testwerkzeug für Webapplikationen, das sich vor allem für Akzeptanztests gut eig- net. Mit dem Firefox Plugin Selenium IDE kann im Browser eine Eingabesequenz aufgezeichnet und danach in einem einfachen Editor bearbeitet werden. Die aufgezeichneten Skripts können dann – auch automatisiert – wieder abgespielt werden. Da Selenium völlig unabhängig von GWT ist, kann es für beliebige Web-Frameworks als Akzeptanztesttool verwendet werden. Der Einsatz von Selenium sollte mit Vorteil erst dann erfolgen, wenn das GUI einigermassen stabil ist und kaum noch Änderun- gen zu erwarten sind. Die Kombination von GWT und Selenium ist eher aufwändig, da zur eindeuti- gen Identifikation von Widgets deren IDs verwendet werden. Leider können die IDs nicht mit dem UIBuilder zugewiesen werden und müssen daher mühsam von Hand im Code gesetzt werden.

8. HTML 5

8.1. Motivation

Dynamische Websites sind Standard, Interaktionen zwischen Usern (Mail, Facebook ,Twitter, etc) sowie schnell reagierende Rich Internet Applications gehören zum guten Ton. Nicht zu vergessen sind Anforderungen an die Sicherheit aber auch an die Barrierefreiheit von Websites.

8.2. Features

HTML5 will in einigen Bereichen HTML an heutige Bedürfnisse anpassen.

- Neue Input-Elemente wie

  Sliders ¡input type=range≫

  Color Pickers ¡input type=color≫

  Telefonnummern ¡input type=tel≫

  Adressen ¡input type=ëmail≫

  Kalender-Date Pickers ¡input type="date≫

  und einige mehr...

- Placeholder Texte sind Texte, die in einem Textfeld angezeigt werden, bevor die Benutzerin zum ersten Mal etwas hineinschreibt. Sie können verwendet werden um Hinweise zu geben, was das Eingabefeld erwartet.
- Autofocus setzt den Cursor beim Anzeigen einer Seite automatisch in ein bestimmtes Inputfeld.
- Geolocation: ein API das es ermöglicht, den Standort des Browsers zu bestimmen.
- Web Workers: das ist ein Standard um Javascript im Hintergrund auszuführen, bzw. mehrere Threads zu starten.
- Offline-Applikationen: Ermöglicht es Webapplikationen auch zu funktionieren, wenn der Browser offline ist.
- Lokaler Speicher: Dem Web-Entwickler steht (mehr oder weniger beschränkt) lokaler Speicher zur Verfügung um Daten auf dem Client zu speichern. Nützlich auch im Zusammenhang mit Offline-Fähigkeit.

  Video: Anzeige von Audio und Video-Dateien ohne Plugins

  Canvas: eine Fläche die beliebig „bemalt“ werden kann.

  Semantische Elemente wie ¡section¿, ¡article¿, ¡nav¿, ¡footer¿... Sie helfen den Text inhaltlich zu strukturieren.

- Microdata: eine Möglichkeit den DOM-Baum semantisch zu annotieren und damit auch ein Schritt in Richtung „semantic web“.

Ist das alles? Ja! Was soll daran neu sein? Ausser dem Standard – nichts. Aber das ist schon eine ganze Menge! Viele der bisherigen Lösungen der oben beschriebenen Features beruhten auf Plugins, umfangreichen Javascript-Libraries oder schlicht auf Hacks. Ein Standard vereinfacht diese Situation

- Features wie Video, lokaler Speicher oder Offline-Support sind nun auf allen HTML5-fähigen Browsern vorhanden, auch dort wo (z.B. aus Sicherheitsgründen) keine Plugins installiert sind.
- Features wie Placeholder Texte und Autofocus sind einheitlich gelöst und müssen nicht für jede Website neu implementiert werden. Gerade das Autofocus-Feature war immer eine mühsame Fallunterscheidung zwischen den verschiedenen Browsertypen um die Eventreihenfolge richtig zu implementieren.
- Die neuen Input-Elemente vereinfachen in vielen Fällen die Validierung der Benutzereingabe. Gerade Email-Adressen entpuppen sich als erstaunlich komplex (siehe z.B. http://www.regular-expressions.info/email.html)
- Zudem ist es nicht mehr nötig diesen Code für jede Website neu als JS-Bibliothek herunterzuladen. Der Code ist im Browser, dort geht er nicht vergessen und ist wahrscheinlich stabiler als stetig wiederholter Skriptcode.

8.3. Mobile Geräte

Viele der Neuerungen sind für herkömmliche Webbrowser tatsächlich nicht atemberaubend. Interessant ist HTML5 allerdings auch für die mobilen Geräte! So unterstützt iPhone viele HTML5 Features, webOS von HP/Palm baut sogar auf Javascript und HTML5 als Basis für die Applikationsentwicklung. Wo liegen die Vorteile für mobile Geräte?

- Mobile Geräte haben oft eine langsamere Verbindung ins Internet. Benutzer sind dankbar für jede JS-Library die nicht heruntergeladen werden muss.
- Verbindungen können unterwegs immer mal wieder abreissen (z.B. in Tunnels). Da ist man froh um lokalen (Zwischen-)Speicher und Offline-Funktionalitäten.
- Geolocations machen auf einem stationären PC der seine IP-Nummer über einen (weit entfernten) Provider bezieht nicht wirklich Sinn.
- Plugins sind für mobile Geräte auch oft ein Tabu – einerseits aus Sicherheitsgründen, andererseits auch wegen der beschränkten Anzeigefläche.
- Die neuen Inputelemente bieten nicht nur eine Validierung an, sondern sie geben dem mobilen Gerät auch einen Hinweis welche Tastatur eingeblendet werden soll. Wird also eine Telefonnummer verlangt, kann das iPhone z.B. eine Tastatur verwenden, die nur Ziffern, *, # und + aufweist.

8.4. Chaos mit Standards

Nein! Das Durcheinander von „Standards“ wird auch in Zukunft bleiben, ja sich vielleicht sogar noch verschärfen, da jetzt auch die mobilen Browser mitmischen werden. Gründe warum es auf absehbare Zeit nicht einfacher werden wird mit HTML umzugehen:

- Rückwärtkompatibilität HTML 5
- Kein Standard - W3C Working Draft.
- Es stehen handfeste Interessen hinter den Produkten. Jeder Browserhersteller möchte sich so gut wie möglich positionieren und den anderen so wenige Chancen wie möglich bieten, eine bessere Position zu erlangen. Als Beispiel hierfür kann der lokale Speicher genommen werden. Es gab Bestrebungen eine Datenbank in die Browser zu integrieren, damit nicht nur Key-Value-Paare persistent gespeichert werden können. Die sogenannte WebDB lehnte sich stark an SQLLite an. So stark, dass man nicht mehr von einem offenen Standard sprechen kann (was die Spezifikation selbst sogar zugibt). Mozilla und Microsoft haben WebDB nicht übernommen.

Teil II.

Arbeitsblätter, Übungen, Code

9. MVP, Navigation

Listing 9.1: EntryPoint

```
package webfr.simple.client;

import webfr.simple.client.presenter.Presenter;
import webfr.simple.client.presenter.WelcomePresenter;
import webfr.simple.client.view.WelcomeView;

import com.google.gwt.user.client.ui.HasWidgets;

public class AppController {

  private Presenter start, current;
  private HasWidgets container;

  public AppController() {
    start = new WelcomePresenter(new WelcomeView(), this);
  }

  public void start(final HasWidgets container) {
    if (container != null) {
      this.container = container;
    }
    showViewOf(start);
  }

  public void showViewOf(Presenter p) {
    current = p;
    current.present(container);
  }

}
```

Listing 9.2: AppController

```
package webfr.simple.client.presenter;

import webfr.simple.client.AppController;
import webfr.simple.client.view.OneDisplay;

import com.google.gwt.event.dom.client.ClickEvent;
import com.google.gwt.event.dom.client.ClickHandler;
import com.google.gwt.user.client.ui.HasWidgets;

public class OnePresenter implements Presenter {

  OneDisplay display;
  int counter = 0;
  AppController controller;

  public OnePresenter(OneDisplay d, AppController ac) {
    display = d;
    controller = ac;
  }

  @Override
  public void bind() {
    // set handler of counter button
    display.setClickHandler(new ClickHandler() {

      @Override
      public void onClick(ClickEvent event) {
        counter++;
        display.setNofClicks("" + counter);
      }
```

```
    });

    // set handler of back button
    display.setBackHandler(new ClickHandler() {

      @Override
      public void onClick(ClickEvent event) {
        controller.start(null);
      }

    });
  }

  @Override
  public void present(HasWidgets container) {
    container.clear();
    container.add(display.asWidget());
  }

}
```

Listing 9.3: OneView

```
package webfr.simple.client.view;

import com.google.gwt.core.client.GWT;
import com.google.gwt.event.dom.client.ClickHandler;
import com.google.gwt.uibinder.client.UiBinder;
import com.google.gwt.uibinder.client.UiField;
import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.Composite;
import com.google.gwt.user.client.ui.Widget;

public class OneView extends Composite implements OneDisplay {

  private static OneViewUiBinder uiBinder = GWT.create(OneViewUiBinder.class);

  interface OneViewUiBinder extends UiBinder<Widget, OneView> {
  }

  public OneView() {
    initWidget(uiBinder.createAndBindUi(this));
  }

  @UiField Button button;
  @UiField Button back;

  public OneView(String firstName) {
    initWidget(uiBinder.createAndBindUi(this));
    button.setText(firstName);
  }

  @Override
  public void setNofClicks(String s) {
    button.setText(s);
  }

  @Override
  public void setClickHandler(ClickHandler h) {
    button.addClickHandler(h);
  }

  @Override
  public void setBackHandler(ClickHandler h) {
    back.addClickHandler(h);
  }

}
```

Listing 9.4: WelcomeView

```
package webfr.simple.client;
```

```
import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.user.client.ui.RootPanel;

/**
 * Entry point classes define <code>onModuleLoad()</code>.
 */
public class Simple implements EntryPoint {
  /**
   * This is the entry point method.
   */
  public void onModuleLoad() {
    AppController appViewer = new AppController();
    RootPanel r = RootPanel.get("appContainer");
    appViewer.start(r);
  }
}
```

Listing 9.5: WelcomeView xml

```
<!DOCTYPE ui:UiBinder SYSTEM "http://dl.google.com/gwt/DTD/xhtml.ent">
<ui:UiBinder xmlns:ui="urn:ui:com.google.gwt.uibinder"
  xmlns:g="urn:import:com.google.gwt.user.client.ui">
  <ui:style>
    .important {
      font-weight: bold;
    }
  </ui:style>
  <g:HTMLPanel>
    <g:Button styleName="{style.important}" ui:field="one" />
    <g:Button styleName="{style.important}" ui:field="two" />
  </g:HTMLPanel>
</ui:UiBinder>
```

Listing 9.6: WelcomePresenter

```
package webfr.simple.client.presenter;

import webfr.simple.client.AppController;
import webfr.simple.client.view.OneView;
import webfr.simple.client.view.WelcomeDisplay;

import com.google.gwt.event.dom.client.ClickEvent;
import com.google.gwt.event.dom.client.ClickHandler;
import com.google.gwt.user.client.Window;
import com.google.gwt.user.client.ui.HasWidgets;

public class WelcomePresenter implements Presenter {

  protected WelcomeDisplay display;
  private AppController controller;

  protected ClickHandler oneHandler = new ClickHandler() {
    @Override public void onClick(ClickEvent event) { doOne(); }
  };
  protected ClickHandler twoHandler = new ClickHandler() {
    @Override public void onClick(ClickEvent event) { doTwo(); }
  };

  public WelcomePresenter(WelcomeDisplay d, AppController ac) {
    this.display = d;
    this.controller = ac;
    bind();
  }

  @Override
  public void present(HasWidgets container) {
    container.clear();
    container.add(display.asWidget());
  }

  @Override
  public void bind() {
```

```
    display.setOneButtonHandler(oneHandler);
    display.setTwoButtonHandler(twoHandler);
  }

  protected void doOne() {
    OnePresenter p = new OnePresenter(new OneView(), controller);
    p.bind();
    controller.showViewOf(p);
  }

  protected void doTwo() {
    Window.alert("Two clicked");
  }

}
```

Listing 9.7: OnePresenter

```
package webfr.simple.client.view;

import com.google.gwt.core.client.GWT;
import com.google.gwt.event.dom.client.ClickEvent;
import com.google.gwt.event.dom.client.ClickHandler;
import com.google.gwt.uibinder.client.UiBinder;
import com.google.gwt.uibinder.client.UiField;
import com.google.gwt.uibinder.client.UiHandler;
import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.Composite;
import com.google.gwt.user.client.ui.Widget;

public class WelcomeView extends Composite implements WelcomeDisplay {

  private static WelcomeViewUiBinder uiBinder = GWT
      .create(WelcomeViewUiBinder.class);

  interface WelcomeViewUiBinder extends UiBinder<Widget, WelcomeView> {
  }

  public WelcomeView() {
    initWidget(uiBinder.createAndBindUi(this));
    one.setText("1");
    two.setText("2");
  }

  @UiField
  Button one;
  @UiField
  Button two;

  private ClickHandler oneclick;

  @UiHandler("one")
  void onClick(ClickEvent e) {
    oneclick.onClick(e);
  }

  @Override
  public void setOneButtonHandler(ClickHandler handler) {
    oneclick = handler;
  }

  @Override
  public void setTwoButtonHandler(ClickHandler handler) {
    // TODO Auto-generated method stub

  }

}
```

10. History Management

Listing 10.1: App Controller

```
package webfr.simple.client;

import webfr.simple.client.presenter.OnePresenter;
import webfr.simple.client.presenter.Presenter;
import webfr.simple.client.presenter.WelcomePresenter;
import webfr.simple.client.view.OneView;
import webfr.simple.client.view.WelcomeView;

import com.google.gwt.event.logical.shared.ValueChangeEvent;
import com.google.gwt.event.logical.shared.ValueChangeHandler;
import com.google.gwt.user.client.History;
import com.google.gwt.user.client.ui.HasWidgets;
import com.google.web.bindery.event.shared.EventBus;
import com.google.web.bindery.event.shared.SimpleEventBus;

public class AppController implements ValueChangeHandler<String> {
  public static final String WELCOME_PAGE = "welcome";
  public static final String ONE_PAGE = "one";

  private Presenter welcome, one, current;
  private int clicks = 0;
  private EventBus eventbus = new SimpleEventBus();
  private HasWidgets container;

  public AppController(HasWidgets container) {
    eventbus = new SimpleEventBus();
    this.container = container;
    bind();
  }

  private void bind() {
    welcome = new WelcomePresenter(new WelcomeView(), this);
    one = new OnePresenter(new OneView(), this);
    current = welcome;
    History.addValueChangeHandler(this);
  }

  public void present() {
    current.present(container);
  }

  public int getClicks() {
    return clicks;
  }

  public int incrementClicks() {
    return ++clicks;
  }

  public Presenter getCurrentPresenter() {
    return current;
  }

  public Presenter getWelcomePresenter() {
    return welcome;
  }

  public Presenter getOnePresenter() {
    return one;
  }

  public HasWidgets getContainer() {
    return container;
  }

```

```
    public EventBus getEventBus() {
      return eventbus;
    }

    @Override
    public void onValueChange(ValueChangeEvent<String> event) {
      String token = event.getValue();
      int qm = token.indexOf('?');
      String parameter = (qm < 0) ? "" : token.substring(qm+1);
      if (qm >= 0) {
        token = token.substring(0, qm);
      }
      Presenter p = null;
      if (token == null || token.isEmpty() || WELCOME_PAGE.equals(token)) {
        p = welcome;
      } else if (ONE_PAGE.equals(token)) {
        p = one;
        try {
          clicks = Integer.parseInt(parameter);
        } catch (NumberFormatException e) {}
      }
      if (p != null) {
        current = p;
        present();
      }
    }

  }
```

Listing 10.2: OneView

```
  package webfr.simple.client.view;

  import com.google.gwt.core.client.GWT;
  import com.google.gwt.event.dom.client.ClickHandler;
  import com.google.gwt.uibinder.client.UiBinder;
  import com.google.gwt.uibinder.client.UiField;
  import com.google.gwt.user.client.ui.Button;
  import com.google.gwt.user.client.ui.Composite;
  import com.google.gwt.user.client.ui.Widget;

  public class OneView extends Composite implements OneDisplay {

    private static OneViewUiBinder uiBinder = GWT.create(OneViewUiBinder.class);

    interface OneViewUiBinder extends UiBinder<Widget, OneView> {
    }

    @UiField Button incButton;
    @UiField Button backButton;

    public OneView() {
      initWidget(uiBinder.createAndBindUi(this));
      backButton.setText("Back to Simple Project entry page");
    }

    @Override
    public void setIncNumberHandler(ClickHandler handler) {
      incButton.addClickHandler(handler);
    }

    @Override
    public void setBackHandler(ClickHandler handler) {
      backButton.addClickHandler(handler);
    }

    @Override
    public void setNumber(int num) {
      incButton.setText(Integer.toString(num));
    }

  }
```

Listing 10.3: WelcomeView

```
package webfr.simple.client.view;

import com.google.gwt.core.client.GWT;
import com.google.gwt.event.dom.client.ClickHandler;
import com.google.gwt.uibinder.client.UiBinder;
import com.google.gwt.uibinder.client.UiField;
import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.Composite;
import com.google.gwt.user.client.ui.Widget;

public class WelcomeView extends Composite implements WelcomeDisplay {

  private static WelcomeViewUiBinder uiBinder = GWT
      .create(WelcomeViewUiBinder.class);

  interface WelcomeViewUiBinder extends UiBinder<Widget, WelcomeView> {
  }

  public WelcomeView() {
    initWidget(uiBinder.createAndBindUi(this));
    one.setText("1");
    two.setText("2");
  }

  @UiField
  Button one;
  @UiField
  Button two;

  @Override
  public void setOneButtonHandler(ClickHandler handler) {
    one.addClickHandler(handler);
  }

  @Override
  public void setTwoButtonHandler(ClickHandler handler) {
    two.addClickHandler(handler);
  }

  @Override
  public void setTwoButtonText(int i) {
    two.setText(Integer.toString(i));
  }

}
```

Listing 10.4: WelcomePresenter

```
package webfr.simple.client.presenter;

import webfr.simple.client.AppController;
import webfr.simple.client.event.IncEvent;
import webfr.simple.client.event.IncEventHandler;
import webfr.simple.client.view.WelcomeDisplay;

import com.google.gwt.event.dom.client.ClickEvent;
import com.google.gwt.event.dom.client.ClickHandler;
import com.google.gwt.user.client.Window;
import com.google.gwt.user.client.ui.HasWidgets;

public class WelcomePresenter implements Presenter {

  protected WelcomeDisplay display;
  private AppController app;

  protected ClickHandler oneHandler = new ClickHandler() {
    @Override public void onClick(ClickEvent event) { doOne(); }
  };
  protected ClickHandler twoHandler = new ClickHandler() {
    @Override public void onClick(ClickEvent event) { doTwo(); }
  };

  public WelcomePresenter(WelcomeDisplay d, final AppController app) {
```

```
    this.display = d;
    this.app = app;
    app.getEventBus().addHandler(IncEvent.TYPE, new IncEventHandler() {

      @Override
      public void onIncCounter() {
        display.setTwoButtonText(app.getClicks());
      }

    });
    bind();
  }

  @Override
  public void present(HasWidgets container) {
    container.clear();
    container.add(display.asWidget());
  }

  @Override
  public void bind() {
    display.setOneButtonHandler(oneHandler);
    display.setTwoButtonHandler(twoHandler);
  }

  protected void doOne() {
    app.getOnePresenter().present(app.getContainer());
  }

  protected void doTwo() {
    Window.alert("Two clicked");
  }

}
```

Listing 10.5: OnePresenter

```
package webfr.simple.client.presenter;

import webfr.simple.client.AppController;
import webfr.simple.client.event.IncEvent;
import webfr.simple.client.event.IncEventHandler;
import webfr.simple.client.view.OneDisplay;

import com.google.gwt.event.dom.client.ClickEvent;
import com.google.gwt.event.dom.client.ClickHandler;
import com.google.gwt.user.client.History;
import com.google.gwt.user.client.ui.HasWidgets;

public class OnePresenter implements Presenter {

  protected OneDisplay display;
  private AppController app;

  protected ClickHandler incHandler = new ClickHandler() {
    @Override public void onClick(ClickEvent event) { doInc(); }
  };
  protected ClickHandler backHandler = new ClickHandler() {
    @Override public void onClick(ClickEvent event) { doBack(); }
  };

  public OnePresenter(OneDisplay d, final AppController app) {
    this.display = d;
    this.app = app;
    this.app.getEventBus().addHandler(IncEvent.TYPE, new IncEventHandler() {

      @Override
      public void onIncCounter() {
        display.setNumber(app.getClicks());
      }

    });
    bind();
  }
```

```
    @Override
    public void bind() {
41    display.setIncNumberHandler(incHandler);
      display.setBackHandler(backHandler);
      display.setNumber(0);
    }

46  @Override
    public void present(HasWidgets container) {
      container.clear();
      container.add(display.asWidget());
      display.setNumber(app.getClicks());
51  }

    private void doInc() {
      app.incrementClicks();
      IncEvent event = new IncEvent();
56    app.getEventBus().fireEvent(event);
      History.newItem("one?" + app.getClicks());
    }

    private void doBack() {
61    History.newItem("welcome");
    }

  }
```

Listing 10.6: IncEvent

```
1 package webfr.simple.client.event;

  import com.google.gwt.event.shared.GwtEvent;

  public class IncEvent extends GwtEvent<IncEventHandler> {
6   public static Type<IncEventHandler> TYPE = new Type<IncEventHandler>();

    @Override
    public Type<IncEventHandler> getAssociatedType() {
      return TYPE;
11  }

    @Override
    protected void dispatch(IncEventHandler handler) {
      handler.onIncCounter();
16  }

  }
```

Listing 10.7: IncEventHandler

```
  package webfr.simple.client.event;
2
  import com.google.gwt.event.shared.EventHandler;

  public interface IncEventHandler extends EventHandler {
    void onIncCounter();
7 }
```

11. RPC

11.1. Client

Listing 11.1: Entry Point

```
package ch.fhnw.webfr.sayt.client;

import java.util.List;

import ch.fhnw.webfr.sayt.shared.Address;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.core.client.GWT;
import com.google.gwt.user.client.Timer;
import com.google.gwt.user.client.Window;
import com.google.gwt.user.client.rpc.AsyncCallback;
import com.google.gwt.user.client.ui.Label;
import com.google.gwt.user.client.ui.RootPanel;

/**
 * Entry point classes define <code>onModuleLoad()</code>.
 */
public class SearchAsYouType implements EntryPoint {

  /**
   * Create a remote service proxy to talk to the server-side service.
   */
  private final SaytServiceAsync saytService = GWT.create(SaytService.class);

  private SearchPanel sp;

  // A keeper of the timer instance in case we need to cancel it
  private Timer timeoutTimer = null;

  // An indicator when the computation should quit
  private boolean abortFlag = false;

  // request counter
  private int requestCount = 0;

  static final int TIMEOUT = 5; // 5 second timeout

  /**
   * This is the entry point method.
   */
  public void onModuleLoad() {
    sp = new SearchPanel(this);

    // Use RootPanel.get() to get the entire body element
    RootPanel.get("gwtContent").add(sp);
  }

  public void sayt(String text) {
    // cancel timer that is already running.
    cancelTimer();

    // Create a timer to abort if the RPC takes too long
    timeoutTimer = new Timer() {
      public void run() {
        Window.alert("Timeout expired.");
        timeoutTimer = null;
        abortFlag = true;
      }
    };

    // (re)Initialize the abort flag and start the timer.
    abortFlag = false;
```

```
63      timeoutTimer.schedule(TIMEOUT * 1000); // timeout is in milliseconds

        // Kick off an RPC
        saytService.findAddressMatchingAny(text, new SearchHandler(
            ++requestCount));
68    }

      class SearchHandler implements AsyncCallback<List<Address>> {
        private int requestNo;

73      public SearchHandler(int count) {
          requestNo = count;
        }

        @Override
78      public void onFailure(Throwable caught) {
          cancelTimer();
          if (isCurrent()) {
            RootPanel err = RootPanel.get("errors");
            err.clear();
83          err.add(new Label(caught.getMessage()));
          }
        }

        @Override
88      public void onSuccess(List<Address> result) {
          cancelTimer();
          if (abortFlag || !isCurrent()) {
            // Timeout already occurred. discard result
            return;
93        }
          sp.replaceAddresses(result);
        }

        private boolean isCurrent() {
98        return requestCount <= requestNo;
        }

      }

103   // Stop the timeout timer if it is running
      private void cancelTimer() {
        if (timeoutTimer != null) {
          timeoutTimer.cancel();
          timeoutTimer = null;
108     }
      }

    }
```

Listing 11.2: Search as You Type Service

```
    package ch.fhnw.webfr.sayt.client;

    import java.util.List;
4
    import ch.fhnw.webfr.sayt.shared.Address;

    import com.google.gwt.user.client.rpc.RemoteService;
    import com.google.gwt.user.client.rpc.RemoteServiceRelativePath;
9
    @RemoteServiceRelativePath("sayt")
    public interface SaytService extends RemoteService {
      List<Address> findAddressMatchingAny(String text);
    }
```

Listing 11.3: Search as You Type Service Async

```
    package ch.fhnw.webfr.sayt.client;
2
    import java.util.List;

    import ch.fhnw.webfr.sayt.shared.Address;
```

```
7 import com.google.gwt.user.client.rpc.AsyncCallback;

  public interface SaytServiceAsync {

    void findAddressMatchingAny(String text, AsyncCallback<List<Address>> callback);
12
  }
```

Listing 11.4: Search Panel

```
  package ch.fhnw.webfr.sayt.client;
2
  import java.util.LinkedList;
  import java.util.List;

  import ch.fhnw.webfr.sayt.shared.Address;
7
  import com.google.gwt.cell.client.AbstractCell;
  import com.google.gwt.core.client.GWT;
  import com.google.gwt.event.dom.client.KeyUpEvent;
  import com.google.gwt.safehtml.shared.SafeHtmlBuilder;
12 import com.google.gwt.uibinder.client.UiBinder;
  import com.google.gwt.uibinder.client.UiField;
  import com.google.gwt.uibinder.client.UiHandler;
  import com.google.gwt.user.cellview.client.CellList;
  import com.google.gwt.user.client.ui.Composite;
17 import com.google.gwt.user.client.ui.Label;
  import com.google.gwt.user.client.ui.TextBox;
  import com.google.gwt.user.client.ui.Widget;

  public class SearchPanel extends Composite {
22
    private SearchAsYouType presenter;

    private static SearchPanelUiBinder uiBinder = GWT.create(SearchPanelUiBinder.class);
    @UiField TextBox textBox;
27  @UiField(provided=true) CellList<Address> addressList = new CellList<Address>(new
        AbstractCell<Address>(){
      @Override
      public void render(Context context, Address address, SafeHtmlBuilder sb) {
        sb.appendEscaped(address.getFirstname() + " " + address.getLastname()
            + ", " + address.getStreet() + " " + address.getStreetnr() + ", "
32          + address.getPlz() + " " + address.getCity());
      }
    });

    interface SearchPanelUiBinder extends UiBinder<Widget, SearchPanel> {
37  }

    public SearchPanel(SearchAsYouType presenter) {
      this.presenter = presenter;
      initWidget(uiBinder.createAndBindUi(this));
42    addressList.setEmptyListWidget(new Label("keine Treffer"));
      addressList.setLoadingIndicator(new Label("Suche..."));
      replaceAddresses(new LinkedList<Address>());
    }

47  public void replaceAddresses(List<Address>addresses) {
      addressList.setRowData(addresses);
    }

    @UiHandler("textBox")
52  void onTextBoxKeyUp(KeyUpEvent event) {
      presenter.sayt(textBox.getText());
    }
  }
```

Listing 11.5: search panel xhtml

```
   <!DOCTYPE ui:UiBinder SYSTEM "http://dl.google.com/gwt/DTD/xhtml.ent">
  <ui:UiBinder xmlns:ui="urn:ui:com.google.gwt.uibinder"
    xmlns:g="urn:import:com.google.gwt.user.client.ui" xmlns:p1="urn:import:com.google.gwt.
        user.cellview.client">
```

```
    <g:HTMLPanel>
      <g:VerticalPanel>
        <g:Label text="Bitte Suchtext eingeben:"/>
        <g:TextBox alignment="LEFT" visibleLength="50" ui:field="textBox"/>
        <g:Label text="Resultate:"/>
        <p1:CellList ui:field="addressList"/>
      </g:VerticalPanel>
    </g:HTMLPanel>
  </ui:UiBinder>
```

11.2. Server

Listing 11.6: AddressProvider

```
package ch.fhnw.webfr.sayt.server;

import java.util.LinkedList;
import java.util.List;

import ch.fhnw.webfr.sayt.shared.Address;

/**
 * Service that to lookup an address from an address list.
 * @author Christoph Denzler
 *
 */
public class AddressProvider {

  /**
   * The singleton instance of the address service
   */
  private static AddressProvider instance;

  /**
   * @return Get the singleton instance of the address service.
   */
  public static AddressProvider getInstance() {
    if (instance == null) {
      instance = new AddressProvider();
    }
    return instance;
  }

  /**
   * List of addresses
   */
  private List<Address> addresses = new LinkedList<Address>();

  /**
   * Hide default constructor
   */
  private AddressProvider() {
    initAddresses();
  }

  /**
   * Initialize the list of addresses with some default values.
   * All addresses are invented.
   */
  private void initAddresses() {
    addresses.add(new Address("Moritz", "Leuenberger", "Impleniaplatz", 1, 3000, "Bern"));
    addresses.add(new Address("Dominque", "Gauthier", "Le Chat-Bott ", 12, 1002, "Genf"));
    addresses.add(new Address("Vreni", "Schmidli", "zwischen den Wegen", 15, 9523, "Neu St.
        Johann"));
    addresses.add(new Address("Moritz", "Twenty", "Bahnhofstrasse", 1, 8001, "Z rich"));
    addresses.add(new Address("Stefan", "Leuenberger", " D hlhlzli ", 153, 3005, "Bern"));
    addresses.add(new Address("Andrea", "Leuenberger", "Obersee", 121, 6300, "Luzern"));
  }

  /**
   * Get a list of addresses which match the given text in either first or last names.
   * @param text a junk of text to match in first and last names of addresses.
   * @return the found addresses or empty array if nothing has been found.
   * @throws InterruptedException
   */
```

```
  public List<Address> lookupPartialName(String text) throws InterruptedException {
    int waittime = 10;
63  System.out.println("partial match on: " + text);
    List<Address> ret = new LinkedList<Address>();

    // not a very intelligent implementation, does not scale.
    if (text != null && !"".equals(text)) {
68    if ("M".equals(text)) {
        waittime = 60000; // wait for one minute
      } else if (text.length() == 1) {
        waittime = 4000;  // wait for four seconds
      }
73    System.out.println(" waiting for " + waittime + "ms.");
      Thread.sleep(waittime);

      text = text.toLowerCase();
      for (Address address : addresses) {
78      String first = address.getFirstname().toLowerCase();
        String last = address.getLastname().toLowerCase();

        if (first.contains(text) || last.contains(text)) {
          ret.add(address);
83      }
      }
    } else {
      for(int i = 0; i < 10 && i < addresses.size(); i++) {
        ret.add(addresses.get(i));
88    }
    }

    return ret;
  }
93 }
```

Listing 11.7: SaytServiceImpl

```
  package ch.fhnw.webfr.sayt.server;
2
  import java.util.List;

  import ch.fhnw.webfr.sayt.client.SaytService;
  import ch.fhnw.webfr.sayt.shared.Address;
7
  import com.google.gwt.user.server.rpc.RemoteServiceServlet;

  public class SaytServiceImpl extends RemoteServiceServlet implements SaytService {

12  @Override
    public List<Address> findAddressMatchingAny(String text) {
      try {
        return AddressProvider.getInstance().lookupPartialName(text);
      } catch (InterruptedException e) {
17      throw new RuntimeException("Ooops, this was an InterruptedException");
      }
    }

  }
```

11.3. Shared

```
   package ch.fhnw.webfr.sayt.shared;

  import java.io.Serializable;
4
  /**
   * Data container for a Swiss address.
   * @author Christoph Denzler
   *
9  */
  public class Address implements Serializable {
    private String lastname;
    private String firstname;
    private String street;
```

```
14    private int streetnr;
      private int plz;
      private String city;

      public Address(String firstname, String lastname, String street, int streetnr, int plz,
          String city) {
19      this.firstname = firstname;
        this.lastname = lastname;
        this.street = street;
        this.streetnr = streetnr;
        this.plz = plz;
24      this.city = city;
      }

      public Address() {
      }
29

      public String getLastname() {
        return lastname;
      }
34

      public String getFirstname() {
        return firstname;
      }
39

      public String getStreet() {
        return street;
      }
44

      public int getStreetnr() {
        return streetnr;
      }
49

      public int getPlz() {
        return plz;
      }
54

      public String getCity() {
        return city;
      }
59
  }
```