Fachhochschule Nordwestschweiz
Hochschule für Technik

Component Architecture for Loosely-coupled Responsive Web Applications

Stefan Florian Röthlisberger

A master thesis submitted to the faculty at the University of Applied Sciences Northwestern Switzerland in partial fulfillment of the requirements for the degree of Master of Science FHNW in Engineering with Specialization in Information and Communication Technologies in the FHNW School of Engineering.

Windisch
7. August 2015

Advisor:
Prof. Martin Kropp

Abstract

The goal of this thesis is to design and implement an architecture for a loosely-coupled web application with a responsive user interface optimized for a large multi-touch wall and for tablets. The result is a web application based on the new Web Component specifications that allows to create loosely-coupled custom HTML elements to encapsulate functionality and reuse them in multiple web applications with HTML Imports. An important aspect driving the architecture of the application was the ability to easily add new components (e.g. layouts and widgets). As a case study, the sprint planning 2 meeting of the agile process was implemented, where the user stories are split up into tasks by the team. In the scenario, one person stands at the wall while all others have a tablet to create tasks for the currently discussed user story. The challenge for the UI was to adapt it from the wall to the much smaller tablet by retaining the same functionality as on the wall. By using media queries to adjust the style of the website and ability to hide parts of it based on the dimensions of the viewport, the web application adapts itself to the available space of the viewport and thus can provide an adequate viewing experience on both device types.

Contents

Chapter 1

Introduction

1.1 Objective

The objective of this thesis is to design and implement a software architecture for a web application based on loosely-coupled components. The web application should be able to run on large multi-touch walls as well as tablets by providing an appropriate viewing experience on both. An important aspect of the web application is to be easily extendable with new components. The sprint planning 2 (SP2) meeting is used as a case study for the architecture and the responsive UI. The collaborative meeting is held with the large multi-touch wall and multiple tablets. The requirements, that were defined at the beginning of the project, can be found in the appendix chapter B or in the project declaration [26].

1.2 Overview

The following chapter provides some background information and information about preliminary work done by me and other related work. Then, a chapter describes the UI design and the challenges faced when adapting an UI designed for a large multi-touch wall to a tablet. The following chapters describe the architecture and implementation of the web application. The application is based on Web Components [9], a new web technology based on four specifications, which are described in a separate chapter. The discussion provides some insight into problems and discusses the work I have done. Finally, the last chapter summarizes the thesis and provides an outlook into possible further developments.

Chapter 2

Background

Large pin boards are still often used for agile project planning instead of digital solutions. The web application I developed for this thesis, called *aWall* (Figure 2.1), is intended to replace the pin board. The system consists of two distinct device types: The wall as a direct replacement for the pin board and tablets for more input-oriented tasks. This presents unique challenges regarding the interaction with the system as a team with the wall or as an individual with the tablet.

2.1 The Agile Development Process

Agile project management promises that a project is finished on time, delivers high quality software, is well constructed and has maintainable code [27]. The core of agile is a set of practices and methodologies around self-organizing teams relying on face-to-face communication [8]. According to a recent study conducted in Switzerland, 70% of all IT companies practice agile software development and 83% of all IT professionals [18]. Agile software development projects are split into short iterations usually lasting between 2-4 weeks. In Scrum these iterations are called *sprints*. The user requirements for the software system are described in *user stories*, a brief description of a feature in everyday language. All *user stories* are stored in a *product backlog* and are maintained by a *product owner*, who represents the customer to the development team and who is in charge of all requirements. Each sprint starts with a *sprint planning meeting*, which is divided in two sub meetings. In the sprint planning 1 (SP1) meeting, the team and the product owner select those user stories which will be implemented in the next iteration. The selected user stories are stored in a list called the *sprint backlog*. In the following SP2 meeting, the team splits the selected user stories from the *sprint backlog* into smaller technical tasks, which describe what the developer has to do to implement the user story. These tasks should not take longer than one workday to complete.

The typical way doing this is to write the user stories and tasks on cards and attach them to a large pin or magnetic board. The setup on the board is called a *taskboard*.

It is typically divided into three columns (Todo, In Progrees, Done) which indicate the progress of the individual tasks. For each user story there is a row in the table. This allows the team to easily see the state and progress of the current sprint. While a physical card wall offers many advantages in respect of ease of use and flexibility, it also has major disadvantages compared to digital solutions. For example, the information represented on the physical taskboard is not saved in an application (e.g. not tied to a bug tracking system) and can not be modified remotely. However, recent studies [15, 20] show that current desktop-based digital solutions can hardly fulfill the requirements concerning agile collaboration.

Large digital multi-touch wall systems have the potential to replace the physical boards and offering even more possibilities, amongst are also support for distributed team collaboration. However as [15] and [20] show, they must provide the natural interaction and ease of use like physical boards. In [20] Mateescu et al. present a concept for a large multi-touch wall-based agile collaboration platform. The agile software development process comprises of different kinds of meetings, which can all be held at the wall. For some of the meetings a direct interaction with the wall, respectively with the artifacts represented at the wall, seems less appropriate. Other, smaller, input devices like individual developers tablets or even smartphones might be more appropriate. These devices allow multiple people to directly and independently interact with the artifacts on the wall and enter or modify data during the meeting more comfortable; still everybody immediately seeing the changes made. In the collaborative SP2 meeting, where everyone helps in creating tasks for the next iteration, tablets can be used to create the tasks in an interconnected web application.

2.2 Previous Work

2.2.1 Agile Technology for Agile Methods (ATAM) [1]

Agile Technology for Agile Methods was an interdisciplinary research project involving multiple institutes at the University of Applied Sciences and Arts Northwestern Switzerland FHNW. Based on an interview study with eleven agile companies about communication and collaboration, new concepts were developed for an agile collaboration platform using large multi-touch walls. Figure 2.1 shows a mockup of the UI concept on 2x2 full-HD monitors forming a 4K display. The UI design of *aWall* is based on these concepts. Note that the UI designs of this project are only for the large multi-touch wall and not for tablets.

[1] http://www.fhnw.ch/technik/imvs/forschung/projekte/si-atam

Figure 2.1: A mockup of the UI concept developed in the Agile Technology for Agile Methods (ATAM) project running on a large multi-touch wall composed of 2x2 displays.

2.2.2 Agile Data Services (ADS)

In my first project Agile Data Services (ADS) [24], I implemented a REST API and a notification mechanism using Server-sent Events based on the ASP.NET Web API [2] framework written in C#. The idea behind this generic REST API is the ability to add arbitrary backends to store the data in third-party issue and bug tracking systems.

2.2.3 Agile Multi-Touch Task Board (AMTTB)

In my second project revolving around the agile taskboard [25], I extended the server application developed in ADS with a JIRA backend and implemented a first prototype of a multi-touch web application using AngularJS [3], jQueryUI [4] and Bootstrap [5]. The library I used in the web application for touch-based interaction like resizability and drag & drop was jQueryUI. The big problem with the library is that it has no support for touch events by itself. Thus I used jQuery UI Touch Punch [6], a small hack to use touch events with the jQueryUI library. Although touch input worked, multi-touch did not. The problem

[2] http://www.asp.net/web-api
[3] https://www.angularjs.org
[4] https://jqueryui.com
[5] http://getbootstrap.com
[6] http://touchpunch.furf.com

lies in the way the hack works. By mapping mouse events to touch events, touch input works. But since a computer only has one mouse (one pointer), using multiple fingers on a multi-touch screen can not work with this hack. Another issue I noticed was that the code became difficult to maintain over time having Javascript and HTML separated and using the MVC pattern with ever growing and nested controllers.

2.3 Related Work

The taskboard is the core component of agile planning. It shows the current progress of the project iteration at a glance and is used in the daily stand-up meetings. A digital tabletop application is introduced for agile planning meetings in [11]. The tabletop only has a single-touch display. Thus only one person can interact with the system at a time and without advanced gestures that involve multiple fingers like pinch-to-zoom. In [23], a cooperative multi-touch taskboard for large displays is proposed. The paper discusses how interactive tabletops and surfaces can be applied to the agile taskboard for the daily stand-up and other meetings and evaluates gestures known from smartphones for big tabletops and group settings. [16] introduces a task assignment system for tabletop computers that consists of two components: The tabletop application and a separate Android companion application. The companion application shows changes made on the tabletop in real-time and offers some limited interaction capabilities.

A multi-surface communication and collaboration platform using a wall, tabletops and tablets for an emergency operations center is presented in [7]. The system is spatially aware of all the devices and people in the room using multiple Kinects. This allows the use of gestures like a flick on a tablet towards the wall to share information. The device types are used for different scenarios: The tablet is used for personal and private interaction with colleagues. Whereas the tabletop is a semi-private workspace for collaboration and the wall is a public information radiator, aggregating multiple data sources. The authors developed native applications for the wall, the tabletop and the tablet. Thus multiple independent applications were developed with different code bases using the same information provided by servers.

One of the most significant challenges with the emergence of all the new devices like smartphones, tablets and phablets in the last years, is device and platform fragmentation. Developing a unique experience for each device would mean to have an application per device type. Consequently, this also means multiple times the work if something needs to be changed. By using a single code base that serves a multitude of devices, a maintenance nightmare can be avoided. For the web, HTML5 and CSS3 bring many advancements to create a single code base for different device and screen sizes. The approach using a single application is called responsive web design (RWD). In [21], the author presents a case study of a mobile-first design process used to create a hybrid

smartphone and tablet application for transaction banking.

What responsive design does not take into account is the proximity of the user. The authors in [28] developed an example implementation of a responsive web application that adjusts the information displayed according to the user's distance to the display.

In [4], a web-based application framework is presented for collaborative visualizations across multiple devices. The framework allows to synchronize UI state of new or legacy web applications with minimal overhead.

Chapter 3

UI Design

Using one application to cover multiple devices with different dimensions presents some challenges:

A. **Device interaction:** How can the two device types be used to conduct collaborative meetings.

B. **Responsive design**

   1. **Retain functionality:** How can the UI be designed to offer the same functionality on the tablet as on the wall.
   2. **Device-specific viewing experience:** How can the UI be designed to offer a device-specific viewing experience when the two device types have significantly different screen resolutions and physical dimensions.
   3. **Little to no scrolling** How can the UI be designed so the whole content of the application is visible without much scrolling.

The following subsections are going to explain the aforementioned challenges in detail and how they were solved in *aWall* using the SP2 meeting as a case study.

3.1 Interaction

The wall's strength is to provide overview and interaction methods like assigning a person to a task or moving a task from 'In Progress' to 'Done'. The *aWall* application on the wall can be operated by multiple people at the same time using the aforementioned interaction methods. Text-input using the virtual or physical keyboard is possible but limited to one person at a time. However, the tablet is much more input-friendly, particularly when it supports pen input with handwriting detection. The SP2 meeting is one of the most collaborative meetings in the agile process with all team members actively participating in the break down of user stories to tasks. By using the multi-touch

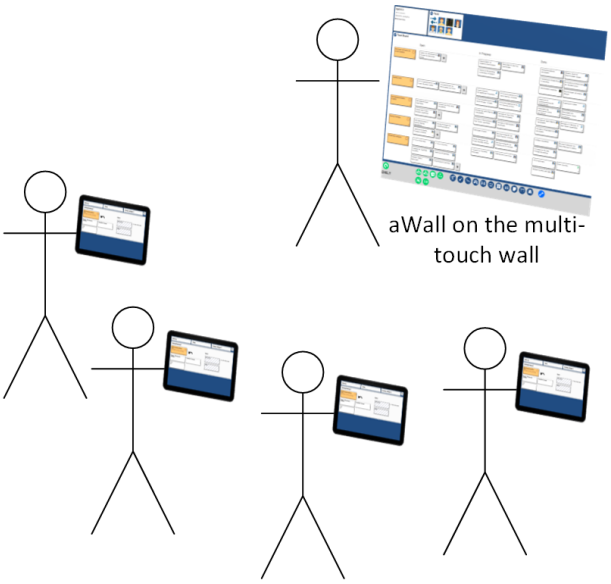


Figure 3.1: The scenario for the sprint planning 2 meeting using the multi-touch wall and tablets.

wall and tablets for the meeting (Figure 3.1), the team gathers around the wall with one person handling the wall and everyone else equipped with a tablet. People with a tablet create the tasks for the currently discussed user story. The tasks they create immediately show up on the wall after being saved as unassigned tasks (right panel in Figure 3.2). Unassigned tasks are specific to *aWall* and represent the paper-based card written by a team member that has not been discussed by the team and thus has not been assigned to a user story yet. The unassigned tasks are discussed and then either assigned to a user story or discarded by the person at the wall.

3.2 Responsive Design

3.2.1 Background

The responsive web design (RWD) [19] approach offers a solution to create one application that provides an optimal viewing experience across a wide variety of different devices with various display resolutions.

RWD builds on the following three cornerstones [19]:

- Relative units: The style and layout definitions should use relative units like percent instead of fixed units like pixel. A pixel on a 1080p tablet display does not have the same physical size as a pixel on a large 4K multi-touch wall system.
- Flexible images: The images are sized in relative units but there may also be multiple versions of the same image. For example, a low resolution image for small-screen devices and a higher resolution image for larger devices.

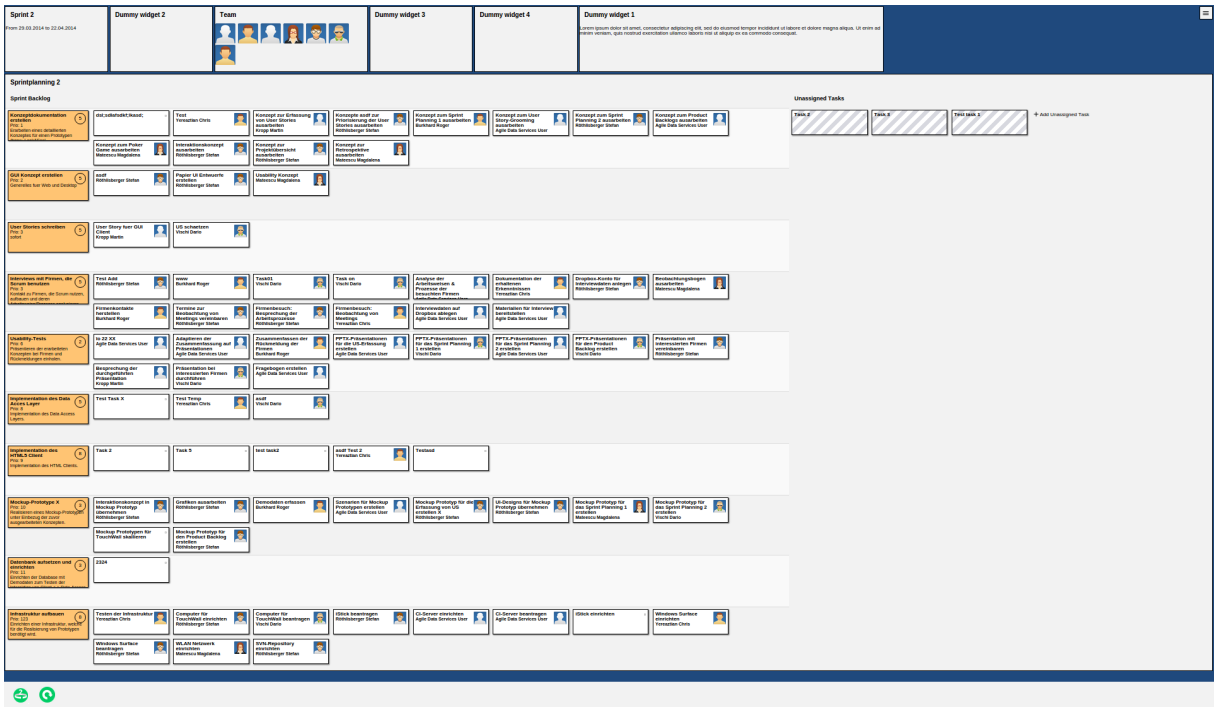


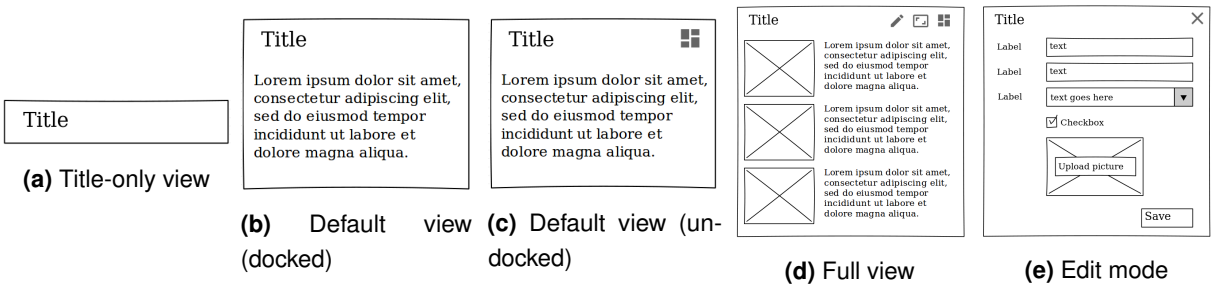
Figure 3.2: UI layout of *aWall* with a main widget for SP2 as seen on the multi-touch wall with a 4K screen resolution.

- Media queries: Media queries are the main cornerstone of RWD. They allow us to define breakpoints with conditional statements in the stylesheet at which different CSS style definitions take effect.

Media queries have been introduced in CSS3 and became a W3C standard in June 2012 [22]. They allow the content of a website to adapt to the conditions of the web browser. The most prominent example of such a condition is the screen resolution. By defining break-points, using for example the width of the viewport, the statement defines the style of a Document Object Model (DOM) element. Let's assume we have a simple element with a title in a `<h1>` element and a text in a `<p>` element. The media query could define that the `<p>` element is hidden if the height of the viewport is less than 500 pixels.

An important concept of RWD is mobile-first [29]. It is an approach to give the mobile site a higher priority and design it with the constraints and capabilities of small-screen devices in mind. The approach is based on the assumption that adding content to a website designed for small-screen devices is easier than removing content from the large desktop layout to fit the smaller screen.

A big difference between *aWall* and a normal website, the basis for all the examples in books about responsive design and the mobile-first approach [19, 29], is how the application is used. Websites on the Internet are content-oriented and we visit them to consume text and images. *aWall* on the other hand is interaction-oriented where the



(a) Title-only view

(b) Default view (docked)

(c) Default view (undocked)

(d) Full view

(e) Edit mode

Figure 3.3: The different views for info-view widgets.

user uses the multi-touch screen to interact with artifacts of the application.

In RWD, the design of a website is optimized to the screen-size. This also means that the mobile site shows only the most important content, because the available space is very small. Additional content shown on the desktop website is hidden and only accessible through menus and functionality is often sacrificed in order to keep it organized and legible. This works in a content-oriented website, but not an interaction-oriented website. Especially when the main goal is to provide the same functionality on the tablet as on the large wall (Challenge B1), but still provide an appropriate viewing experience on the device (Challenge B2). The biggest challenge therefore was to bring all the information and functionality from the wall to the much smaller tablet.

3.2.2 General Layout

The agile process has multiple collaborative meetings that deal with different aspects of the project. The UI of *aWall* in Figure 3.2 is built around workspaces configured with independent widgets that can be used by several workspaces. For each meeting there is a workspace configured with the most appropriate widgets. Each workspace consists of a main widget that takes most of the available space, numerous independent widgets in a panel at the top of the screen, called info-view, and the workspace-menu with options to configure the workspace in the top-right corner.

The main widget occupies the largest part of the UI and is the main working area. For each meeting of the agile process there is a specialized main widget that shows exactly what is needed and offers tailored interaction methods. Supplemental interaction with additional artifacts is provided by the widgets in the info-view. Each main widget is responsible for the responsive behavior of its content to show the appropriate amount of information depending on the available space.

The info-view is the horizontal panel at the top with the smaller widgets. The widgets in the info-view can have different widths from standard width to multiple-times the standard-width. The number of widgets displayed is calculated by the application using a defined minimal width for info-view widgets. The broader the viewport, the more widgets can be displayed. Info-view widgets can be enabled or disabled. If a currently

disabled widget is enabled and the info-view is full, the widget in the last position is replaced with the newly enabled widget. This ensures that only the calculated number of widgets are displayed which ensures that the widgets always have enough space. Disabled widgets can be accessed by the workspace-menu, where each widget can be enabled or disabled.

The widgets in Figure 3.2 are in a docked position but can be undocked by dragging them out of the info-view. When a widget is dragged out of the info-view (undocked) the space is closed whereby all widgets to the right move one position to the left. It can now be moved around freely and resized by either using the mouse or the pinch-to-zoom gesture on a multi-touch display. The floating widget can be docked in any position, even if the position is already occupied. Is a position occupied by another widget, the position is freed up with all widgets to the right moving one position to the right. Is the application run on a large screen and all visible widgets do not take up the available horizontal space, a widget dropped in a position to the far right is docked in the first position to the left of the last docked widget. For instance if room for 8 widgets is available, but only 4 widgets are present (3 docked, 1 undocked). Then the undocked widget that is dropped into position 7, is docked in position 4.

The widgets can have multiple views, as depicted in Figure 3.3, for different sizes when being resized. In the docked position the widgets show the default view (Figure 3.3b) on larger screens. On smaller screens, like a tablet, only the title of the widget is shown (Figure 3.3a), henceforward called title-only view. The default size with the same functionality as on larger screens is shown when it is undocked (Figure 3.3c). Besides the title-only and the default view, a widget can offer an additional full view (Figure 3.3d) and an edit mode (Figure 3.3e). The full view is intended to show more information on the widget's subject or offer additional interaction functionality. It is triggered by specifying the breakpoints for the width and height. Once the widget has exceeded both breakpoints while resizing, the full view is shown. Once a widget is in the undocked state, a small icon in the top-right corner appears (see Figure 3.3c), that docks the widget immediately. When a widget has been resized, even when it has not reached its full view, a 'resize to default' icon is shown, as depicted in Figure 3.3d as the second icon from the right. If a widget has an edit mode, an 'edit' icon is displayed in the top-right corner of the widget (leftmost icon in Figure 3.3d). Once in edit mode, a close button is displayed in the top-right corner (Figure 3.3e) to return to the previous view.

The workspace-menu is the small button in the top right corner. It allows to control which widgets are enabled and disabled with toggles and to save or reset the current workspace configuration (arrangement and visibility of the widgets). On small screens, the workspace-menu also contains links to switch to other workspaces. On larger screens, a horizontal bar is displayed at the bottom of the screen with icons to switch to other workspaces as seen in Figure 3.2.

(a) Overview **(b)** Detailed view

Figure 3.4: The two views of the SP2 user interface for smaller displays.

3.2.3 Sprint Planning 2 Workspace on a Tablet

The multi-touch wall and the tablets have significantly different screen resolutions and physical dimensions. An important goal driving the design was the ability to see everything without much scrolling (Challenge B3).

A lot of vertical space is used up by the info-view widgets when using the default view as on the wall. They offer auxiliary functionality, but it is not absolutely essential that they are completely visible the whole time in their default view. One may argue whether the information and functionality of the info-view widgets is really needed for the tasks a user conducts with the tablet in the scenario. Nonetheless, challenge B1 requires that the functionality should at least be accessible somehow on the tablet. Thus, I created the aforementioned title-only view (Figure 3.3a). That way, the widgets are still in the same place and visible, but use much less space. A downside is that they have to be dragged out of the info-view to be usable. Once the widget is undocked, it enlarges to the default size and it can be used like on the wall.

In the collaborative scenario of the SP2 meeting, the wall and the tablet serve different needs, thus the viewing experience is different (Challenge B2). The wall's duty is to give the whole team the overview of the sprint, while the tablet's focus lies with the individual to create new tasks for the currently discussed user story.

The main widget for the SP2 meeting, as depicted in Figure 3.2, shows the *sprint backlog* on the left and a panel on the right for unassigned tasks. On the wall, everything including the *sprint backlog* and the unassigned tasks can be shown simultaneously. The tablet does not have enough space to show all that information. While the complete *sprint backlog* is important on the wall, it is not relevant for creating a task for a user story. To retain functionality on the tablet (Challenge B1), I split the widget into two views (Figure 3.4): The first view (Figure 3.4a) shows all the user stories of the *sprint backlog* in a grid. When the user clicks, respectively selects a user story, the second view is shown, called detail view (Figure 3.4b). There, the selected user story is displayed with

all its assigned tasks on the left side and the list of unassigned tasks on the right side. The back-arrow to the right of the user story brings the user back to the overview.

Chapter 4

Architecture

4.1 System Overview

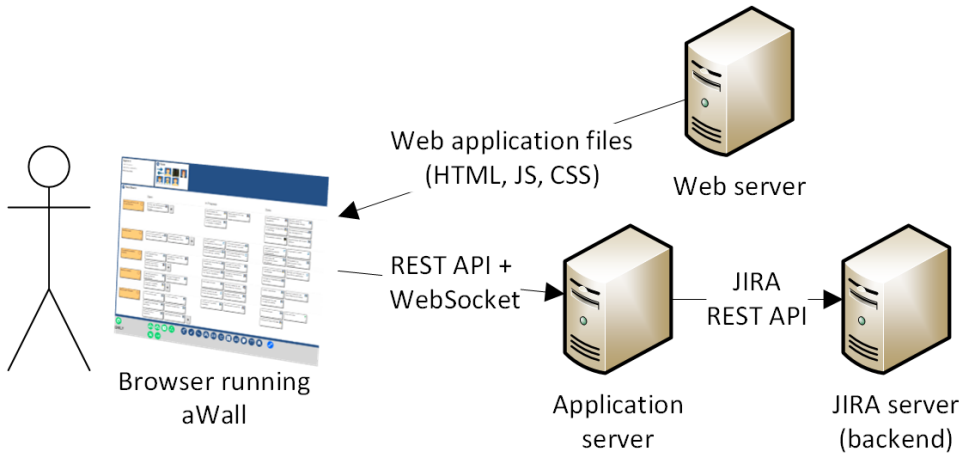


Figure 4.1: System overview.

The whole system consists of four components, as depicted in Figure 4.1:

- *aWall* web application: The *aWall* application is a single-page application (SPA) built using HTML, Javascript and CSS.
- A web server: Delivers the *aWall* web application files to clients.
- Application server: Provides a REST API for *aWall* to retrieve the data about the sprints, user stories, tasks and team members. It also provides a WebSocket endpoint for real-time updates.
- A data backend: The data used by the REST API is provided by a backend. Currently, the system uses JIRA [2] as a backend. JIRA is a bug, issue and project management software used by many software development teams around the world.

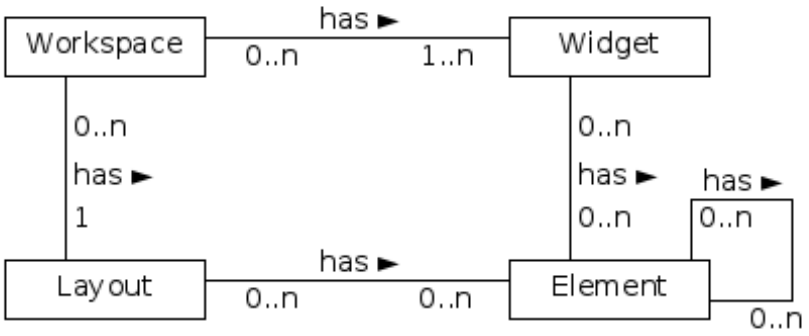


Figure 4.2: Building blocks of the application structure

aWall is a SPA, that means that the website is not rendered by an application server and then sent to the client, but is rendered completely on the client. It also means that the website does not reload, even when another page of the website is loaded. The resources like images and scripts are loaded dynamically when needed and added to the application at runtime. The information about sprints, user stories and tasks is retrieved from the application server over the REST API. The data is delivered in a light-weight format called JavaScript Object Notation (JSON).

4.2 Application Structure

The application is built using the following four building blocks (Figure 4.2):

- **Element**: The elements are independent custom HTML elements that may be used by layouts and by widgets. An element may depend on another element.
- **Widget**: A widget is an independent component that may be used by different workspaces. A widget must not depend on another widget. The widgets are the sole elements that provide interaction and content.
- **Layout**: The layout defines how the widgets are arranged and how many widgets are shown, whereby at least the main widget must be visible. The layout is independent of the workspace and may be used by different workspaces.
- **Workspace**: A workspace may be defined for each meeting of the agile process and defines the layout and the widgets to use. The workspace is the configuration using a combination of the aforementioned layout and widgets.

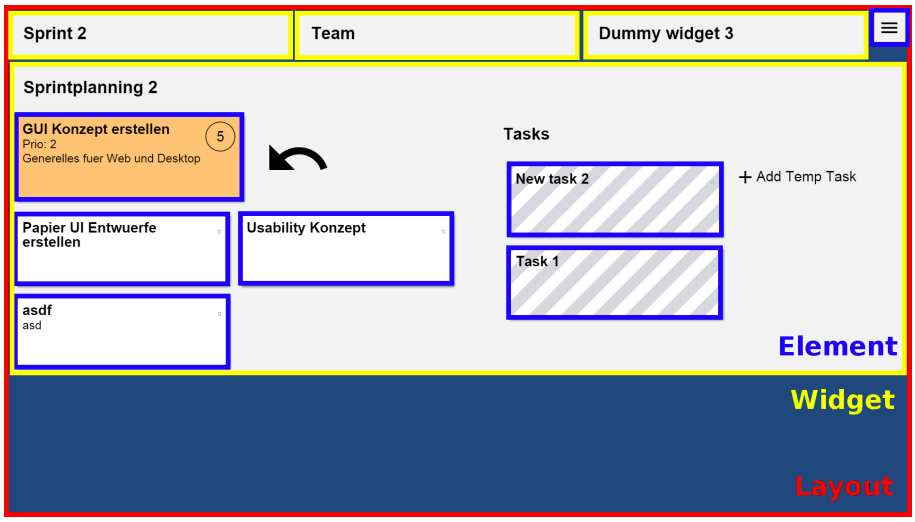


Figure 4.3: Structure of a workspace using the SP2 workspace as an example.

4.2.1 Workspace Structure

The workspace is the configuration using a layout and one or more widgets. A workspace, as shown in Figure 4.3, must have one layout and at least one widget, the main widget. Depending on the layout, the workspace may have additional widgets to supplement the workspace. The layouts and the widgets may use elements.

Figure 4.3 shows the SP2 workspace as seen on a smaller screen like a tablet. The workspace shows three smaller widgets in the top panel, the info-view, and the main widget in the center. The main widget uses multiple elements that are defined in the global space like the user story and task cards, the blue-framed elements, that can be used by other widgets. The layout also uses a global element, the workspace-menu in the top-right corner. Thus, the workspace-menu can be reused in other layouts.

4.2.2 File Structure

For each building block there is a directory under the *app* directory. The workspaces are defined in JSON files in the *workspace* directory. The layouts reside in the directory *layouts* and are defined directly in HTML files. All widgets have a directory for themselves in *widgets*. In the widget's directory there is an *index.html* in which the widget's views are defined. The *widget.json* file contains additional data like the name of the widget and a description. The elements are defined in HTML files in the directory *elements*. The following custom element prefixes are used in the *aWall* web application:

- **ads-***: Elements that provide access to the ADS REST API and the WebSocket endpoint for real-time change notifications.
- **app-***: Global elements that are part of the *aWall* framework.

- **awall-***: Elements specific to the *aWall* application like user story and task cards.
- **core-***: Polymer 0.5 elements.
- **paper-***: Polymer paper-style elements.

The starting point of the application is the *index.html* file in the *app* directory. It has all the imports for the third-party libraries that reside in the *bower_components* directory and the list of all workspace configurations. The *core.js* file contains the core of the application to load all the workspaces on demand and an abstraction for the touch-library.

```
/app/
- bower_components/
- elements/
  - ads-rest.html
  - app-layout.html
  - awall-taskcard.html
- images/
- layouts/
  - single.html
  - workspace.html
- widgets/
  - sprintinfo/
    - index.html
    - widget.json
  - sprintplanning2/
    - index.html
    - widget.json
- workspaces/
  - projectselect.json
  - sprintplanning2.json
- core.js
- index.html
- main.css
```

Listing 4.1: The application's file structure

4.3 Configuration

The workspaces that make up the web application are configured in JSON files, as seen in Listing 4.2. Apart from a name and the description for the workspace, a path, a layout

and widgets must be defined.

```
{
  "name": "Sprint Planning 2 Workspace",
  "description": "Plan a sprint collaboratively.",
  "path": "/project/:projectId/sprint/:sprintId/sp2",
  "layout": "workspace",
  "widgets": {
    "sprintplanning2": {
      "position": "main",
      "hidden": false
    },
    "sprintinfo": {
      "position": 1,
      "width": 1,
      "hidden": false
    },
    "teamview": {
      "position": 2,
      "width": 2,
      "hidden": false
    }
  }
}
```

Listing 4.2: Configuration for the SP2 workspace.

The `path` attribute in the configuration represents the fragment identifier, the part of the URL after the hash character `#`. The fragment identifier refers to a resource that is ancillary to another primary resource, the Uniform Resource Locator (URL). In a SPA, the URL is the path to the web application itself and the fragment identifier represents the state of the application respectively the current website. The path attribute in the configuration is composed of fixed words and words prefixed with a semicolon, the variables. As Listing 4.2 shows, the SP2 workspace has two variables, `projectId` and `sprintId`. The two variables are mapped to real IDs at runtime and are replaced by values like `projectId = "ATBD"` and `sprintId = "10204"`, resulting in the string `"/project/ATBD/sprint/10204/sp2"`. With those two variables, the application can fetch the appropriate information from the server required for the given workspace.

A workspace must define which layout to use. It defines which visual elements (e.g. main widget, info-view, workspace-menu) are displayed and how they are arranged. The *aWall* application currently has two layouts; One is for simple selection pages, for example selecting the sprint which only has a main widget. The other is the workspace layout as seen in Figure 3.2 with the info-view at the top with all the additional widgets and the main widget in the center of the layout. The `widgets` attribute defines all the widgets the workspace has and each widget's configuration in the layout. Each widget defines its position, its width and whether it is hidden by default. A workspace must

have at least one widget, the main widget which has the positional value of `"main"`.

4.4 Real-time Communication

To make the application more responsive in terms of data by offering real-time change notifications, the application server also provides a WebSocket [10] endpoint. WebSocket is bi-directional communication protocol designed to be implemented by web browsers and web servers. It allows both participants (client and server) to send asynchronous messages for real-time communication. Once the client has established the connection, it is persistent until closed explicitly. The protocol I designed for the WebSocket protocol and the application server allows the web application not only to recognize changes made by itself, but also changes made by other instances of the web application on different devices no matter where they are. This works because all instances of the web application establish a WebSocket connection to the application server when started. If, for example a task has been modified, the task is saved by the application server in the backend and then the information about the change is propagated to all clients over the WebSocket connection. When the web application receives an update, it checks whether the current workspace uses this task. If that is the case, the task is updated in-place without reloading the page. When a task has been created for a user story, it will pop-up on all connected devices in real-time.

4.4.1 Message Format

The WebSocket endpoint uses JSON objects to communicate. The format is shown in Listing 4.3 and has three top-level attributes:

- `action`: The name of the action or event that the message represents (e.g. Create, Updated).
- `resourceType`: The name of the resource respectively Data Transfer Object (DTO) (e.g. Task or UserStory).
- `data`: The object with the data. Can be an object, an array or even null depending on the action or event.

```
1 {
2   "action": "CreateTemp",
3   "resourceType": "Task",
4   "data": <!--object, array or null-->
5 }
```

Listing 4.3: The format of all messages used by the WebSocket endpoint.

4.4.2 Actions and Events

The protocol has two types of messages: Actions, that always originate from a client and events, that are triggered by actions via WebSocket or by the HTTP methods `PUT`, `POST` and `DELETE` sent to the ADS REST API. Except for the `GetAllTemp` action/event, all events are propagated to all connected clients. Table 4.1 lists all actions and events with their respective source and how they are propagated.

Source	Action	Triggers Event	Type of Event
WebSocket	`GetAllTemp`	`GetAllTemp`	Unicast
	`CreateTemp`	`CreatedTemp`	Broadcast
	`PersistTemp`	`DeletedTemp, Created`	Broadcast
	`DeleteTemp`	`DeletedTemp`	Broadcast
REST API	`POST /api/...`	`Created`	Broadcast
	`DELETE /api/...`	`Deleted`	Broadcast
	`PUT /api/...`	`Updated`	Broadcast

Table 4.1: Protocol actions and the events triggered.

The `GetAllTemp` action fetches all unassigned tasks. It is usually the first message sent. Unassigned tasks created once connected to the server are retrieved asynchronously with `CreatedTemp` event messages. The `PersistTemp` action saves the unassigned task in the backend and issues a `DeletedTemp` event, so all the clients remove the task from the list of unassigned tasks. It also triggers the `Created` event after the task has been successfully saved in the backend (Listing 4.4).

```
1 handlePersistTemp(dto):
2   if taskBackend.create(dto):
3     handleDeleteTemp(dto)
4     broadcast("Created", dto)
5
6 handleDeleteTemp(dto):
7   if tempTasks.remove(dto):
8     broadcast("DeletedTemp", dto)
```

Listing 4.4: Simplified pseudo code for the `PersistTemp` action.

Chapter 5

Web Components

aWall uses Web Components [9], a set of four standards currently being produced as a W3C specification. Those four specifications offer the following features that can be used together or individually:

- The `<template>` element [17]: Declare inactive DOM-trees that can be reused to instantiate custom elements.
- Shadow DOM [13]: Encapsulate functionality and style in DOM-trees. Normally CSS definitions are global no matter where they are defined. By encapsulating the style definitions in a Shadow DOM, they become local and scoped and do not affect any other DOM elements outside of the Shadow DOM.
- Custom Elements [12] Define and use new HTML elements. That ranges from simple format-elements to complex elements with functionality like the HTML5 `<video>` element.
- HTML Imports [14]: Include and reuse HTML documents. Allows to define custom elements in a separate file, import it and then use those custom elements. Similar to how CSS and Javascript files can be imported.

The *aWall* application is composed of nested custom HTML elements and does not follow the more traditional MVC pattern. This approach goes more in the direction of component-based software engineering that emphasizes the separation of concerns. The custom elements can be defined in separate HTML documents and be reused in multiple web applications by importing the document. This also means that the elements can be loosely-coupled and independent. Even the access to the REST API is encapsulated in an element. To use the REST API, another element only needs to instantiate the appropriate custom element by inserting the corresponding HTML tag into its own HTML.

5.1 Templates

Templates are inactive, reusable DOM-trees that must be cloned and inserted into the DOM-tree by Javascript code. To define a template, the `<template>` element is used. Everything inside the element is inactive: Scripts do not run, styles are not applied, HTML is not visible, images are not loaded and video and audio is not played. The content of the template is not in the DOM. This means that the content of the template can not be queried by functions like `getElementById()` or `querySelector()`.

```
1 <template id="tmpl">
2   Template instances:
3 </template>
4
5 <div id="target"></div>
```

Listing 5.1: `<template>` element definition.

```
1 var numOfClones = 0;
2
3 function cloneTemplate() {
4   var target = document.querySelector("#target");
5   var template = document.querySelector("#tmpl");
6   var clone = document.importNode(template.content, true);
7   clone.textContent += ++numOfClones;
8   target.appendChild(clone);
9 }
```

Listing 5.2: Javascript code to clone a template.

Listing 5.1 defines a simple template that just contains a string. The `<div>` element on line 5 is the element we use to append the content of the template as a child node. The Javascript code necessary just to clone the template is shown in Listing 5.2. The basic steps required are querying the `<template>` element and the target (line 4 and 5), then clone the template's content (line 6). In the example, we just append the number of clones from the variable `numOfClones` to the string of the cloned template on line 7 and then append the cloned template to the target element on line 8.

```
<template id="tmpl">
  <style>
    * { font-family: Georgia, Times, "Times New Roman", serif; }
    h2 { color: green; }
  </style>
  <script>alert("Hello world!")</script>
  <h2>Content of a template without Shadow DOM</h2>
</template>
```

Listing 5.3: Template with style and code, but without Shadow DOM encapsulation.

The DOM-tree in the template element in Listing 5.3 shows an extended template with styles and script. Neither the script is executed nor are the style definitions applied as long as the template is not cloned. Once the template is cloned, the style definitions are applied to the whole document. Every element now has a serif font style and all `<h2>` elements became green. The problem with the style definitions is that they are always global, no matter where they are defined. When defining style definitions inside a template, we do not want the style to be applied for every other element in the document. This encapsulation problem is solved by using Shadow DOM.

5.2 Shadow DOM

Shadow DOM encapsulates styles and HTML inside of the `<template>` element. Thus, the definitions inside a Shadow DOM do not affect or break any other elements outside the Shadow DOM. Shadow DOM has already been used by the browser vendors. Examples are: `<input type="date">` as shown in Figure 5.2, or the `<video>` element. Both elements have extended functionality hidden away by a single HTML tag.

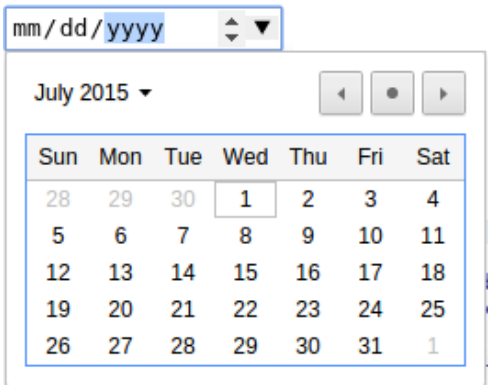


Figure 5.1: The `<input type="date">` element's functionality.

The input `<input type="date">` element shown in Figure 5.1 for example lets the user select the date using different means of input. The date can be set using the arrow-keys on the keyboard, by entering the date directly, by using the up and down arrow in the input field or by clicking on the drop-down arrow on the right and selecting the date in the calendar that opens.

Figure 5.2: Shadow Root of the `<input type="date">` element as seen with the Chrome Developer Tools (can be enabled in Settings -> General -> Show user agent shadow DOM).

```
1 <div id="host">Hello world!</div>
2
3 <script>
4   var host = document.querySelector("#host");
5   var root = host.createShadowRoot();
6   root.textContent = "Overwrite from shadow root";
7 </script>
```

Listing 5.4: Shadow root overwriting the content of the host element.

A shadow root overwrites the content of the host element. The code in Listing 5.4 shows 'Hello World!' to the user until the code, beginning in line 3, is executed. There, on line 5, a shadow root is created in the host element. The text content of the `<div>` element is then replaced with the string 'Overwrite from shadow root' on line 6. The browser now displays the new string instead of 'Hello World!'. Figure 5.3 shows how the DOM-tree looks after the code in Listing 5.4 has been executed.

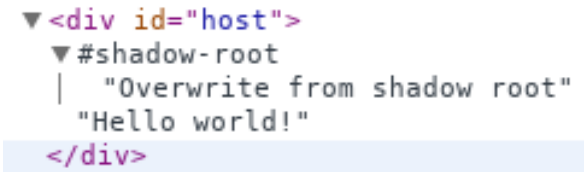


Figure 5.3: The host element with the shadow root as seen with the Chrome Developer Tools.

5.2.1 Insertion Points

```
1  <div id="host">Hello world!</div>
2
3  <template id="tmpl">
4    Content from the host: <content></content>
5  </template>
6
7  <script>
8    var host = document.querySelector("#host");
9    var tmpl = document.querySelector("#tmpl");
10   var root = host.createShadowRoot();
11   var clone = document.importNode(tmpl.content, true);
12   root.appendChild(clone);
13 </script>
```

Listing 5.5: A `<template>` element using a Shadow DOM insertion point.

The example in Listing 5.5 uses the `<content>` element tag on line 4, which is a so called insertion point. The Javascript code again creates a shadow root on the host element, then clones the template and finally appends the cloned template to the shadow root of the host element. The shadow root replaces the host element's content as usual, but the `<content>` element ensures that the content of the host element, the `<div>` element on line 1, is displayed in place of the insertion point in the template. Once the script is executed, the browser displays the text `"Content from the host: Hello World!"`.

The Figure 5.4 shows on the left side the document tree and the shadow tree, and on the right-hand side the composed tree. The shadow root node in the shadow tree is placed into the position of the shadow host node in the document tree. The shadow root node itself has two child nodes, of which one has an insertion point. The two child nodes of the document tree are inserted in place of the insertion point in the composed tree. In the end, the parent node of the insertion point now has its child node from the shadow tree plus the two nodes of the shadow host element of the document tree.

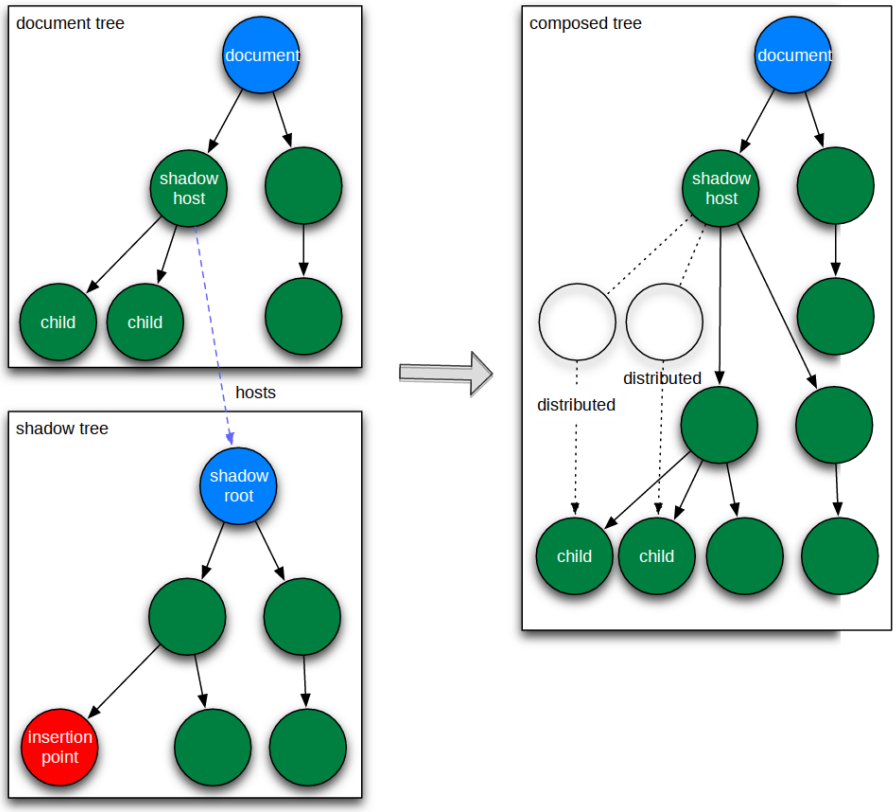


Figure 5.4: How insertion points work. (Source: http://www.w3.org/TR/shadow-dom/#insertion-points)

```
1  <div id="host">
2    <span>'Hello world' in span element</span>
3    <h2>h2 title in the host element</h2>
4    <p>Lorem ipsum ...</p>
5  </div>
6
7  <template id="tmpl">
8    <h2>Shadow DOM using insertion points</h2>
9    <content select="p"></content>
10   <content select="h2"></content>
11   <content select="*"></content>
12 </template>
```

Listing 5.6: A template with multiple insertion points using selectors.

The HTML code in Listing 5.6 shows a host element with multiple elements on lines 1 to 5. The `<template>` element beginning on line 7 has its own child element with the `<h2>` element on line 8 and three insertion points. The first insertion point on line 9 selects all `<p>` elements of the host, the second on line 10 all `<h2>` elements and the third on line 11, the wildcard selector, selects all elements of the host that have not been selected yet. By applying some styles to the code, the result of this template with the

Figure 5.5: The result of the HTML code in Listing 5.6.

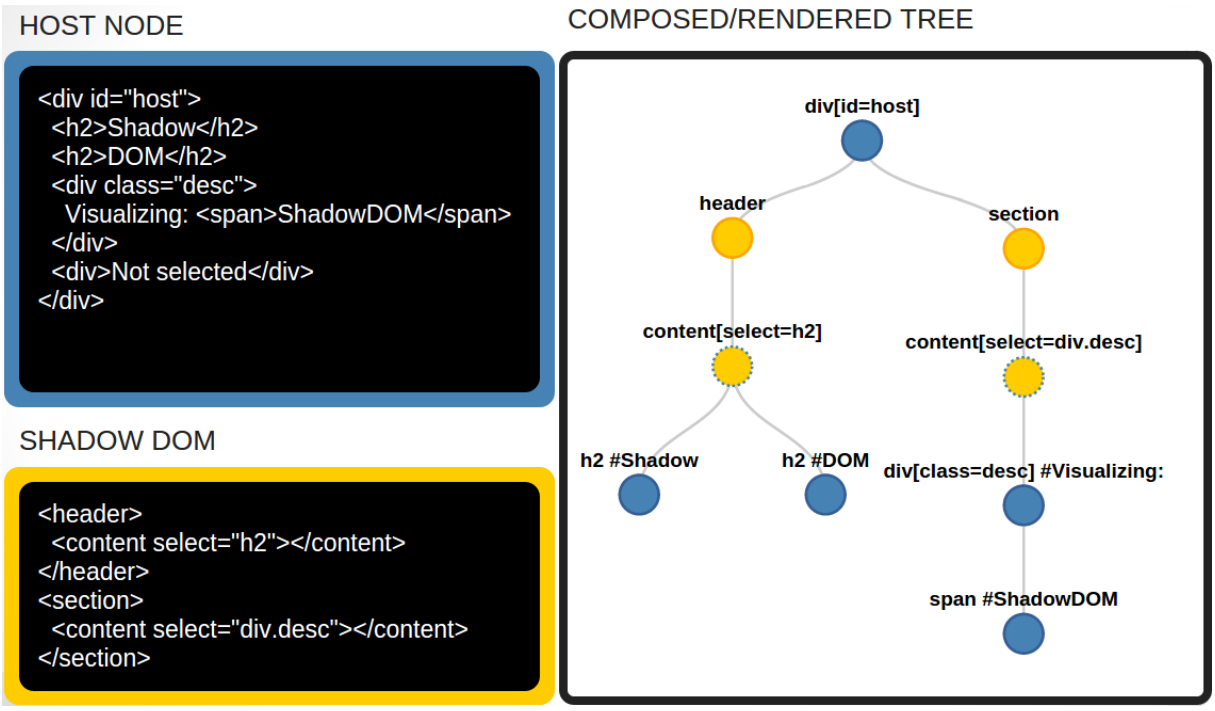


Figure 5.6: Screenshot of an insertion point example using the ShadowDOM Visualizer.

insertion points is depicted in Figure 5.5. Note the element order of the host's elements versus the order in the screenshot and the possibility of giving the elements different styles depending on whether they originated from the host or from the template (shown by the `<h2>` elements in Figure 5.5).

A very useful tool to visualize how Shadow DOM renders the insertion points is the ShadowDOM Visualizer [1] (Figure 5.6).

[1] https://html5-demos.appspot.com/shadowdom-visualizer

5.3 Custom Elements

The custom elements specification allows to create new HTML elements or extend native HTML elements. As with the `<video>` tag, we can create HTML elements that hide and encapsulate functionality behind a single HTML tag. The only restriction of custom elements applies to naming. To differentiate between native HTML elements and custom elements, all custom elements must contain a dash '-' in their name.

Custom elements do not have to be registered before the browser reads the HTML tag. All elements that browser does not recognize are hidden and are categorized as *unresolved elements*. Once the Javascript code registering a custom element is parsed and executed, the unresolved element is upgraded and becomes visible.

Creating a Custom Element

Creating a new custom element is simple: Just register the new HTML tag, `<my-element >` in the example, as shown in Listing 5.7.

```
1 document.registerElement("my-element");
```

Listing 5.7: Registering a custom element.

Extending a Native Element

Extending a native element is also pretty straight-forward. The code in Listing 5.8 has an object as the second argument to `registerElement()` with additional information relating to the prototype of the element, that loosely translates to a base class in object oriented programming, and the tag name of the element we extend. The usage of this element is slightly different. Instead of using `<my-button>`, we have to use `<button is="my-button"></button>`.

```
1 document.registerElement("my-button", {
2   prototype: Object.create(HTMLButtonElement.prototype),
3   extends: "button"
4 });
```

Listing 5.8: Registering a custom element that extends a native element.

Extending a Custom Element

Extending another custom element is very similar to extending a native element. For the prototype, we give it the one of the `<my-element>` element. The extended element can then be used with `<my-element-extended>` (Listing 5.9).

```
var MyElementProto = Object.create(HTMLElement.prototype);
...
document.registerElement("my-element-extended", {
  prototype: Object.create(MyElementProto.prototype),
  extends: "my-element"
});
```

Listing 5.9: Registering a custom element that extends a another custom element.

Lifecycle callbacks

The callbacks offered by custom elements, as listed in Table 5.1, can be used to have code executed, for instance, once the custom element has been attached to the document with `attachedCallback`.

`createdCallback`	An instance of the element has been created
`attachedCallback`	An instance has been inserted into the document
`detachedCallback`	An instance has been removed from the document

Table 5.1: Life cycle callbacks of custom elements.

```
var MyElementProto = Object.create(HTMLElement.prototype);
MyElementProto.createdCallback = function() {
  this.innerHTML = "Lorem ipsum";
};
document.registerElement("my-element", {prototype: MyElementProto});
```

Listing 5.10: Registering an custom element callback.

5.4 HTML Imports

HTML Imports provides a simple method to import another HTML document with a single HTML tag like the `<script src="lib.js">` element does for Javascript files. Before HTML Imports came along, it was possible to import other HTML files using an iframe or fetch it with an AJAX call and then append it to the document. Both of these methods are more of a hack and have some drawbacks. With this new specification, importing another HTML document is much easier. As with script and stylesheet imports, it is a declarative definition in the document. It uses the `<link>` element with the attribute `rel="import"` (Listing 5.11).

```
<head>
  ...
  <link rel="import" href="my-element.html">
</head>
```

Listing 5.11: HTML import for the file *my-element.html*.

5.5 Putting it all together

This section shows how all these four specifications of Web Components can be used together.

```
1  <template id="my-element-tmpl">
2    <style>...</style>
3    <h2>HTML imported custom element using shadow DOM</h2>
4    <p>I'm the custom element 'my-element'</p>
5  </template>
6
7  <script>
8    var MyElementProto = Object.create(HTMLElement.prototype);
9    MyElementProto.createdCallback = function () {
10     var tmpl = document.querySelector("#my-element-tmpl");
11     var clone = document.importNode(tmpl.content, true);
12     var root = this.createShadowRoot().appendChild(clone);
13   };
14   document.registerElement("my-element", {prototype: MyElementProto});
15 </script>
```

Listing 5.12: The content of *my-element.html*.

The Listing 5.12 shows the content of *my-element.html*. In this HTML file, the element `<my-element>` is defined by using Custom Elements, Templates and Shadow DOM. The template is defined on lines 1 to 5. The prototype for the new custom element is created by using `HTMLElement.prototype`, the base of HTML elements. On this prototype, we now register the `createdCallback`, where the template is queried on line 10 and cloned on line 11. Then on line 12, a shadow root is created on the `this` variable, which is the prototype of the new custom element. Finally, `<my-element>` is registered using the prototype we created with the callback.

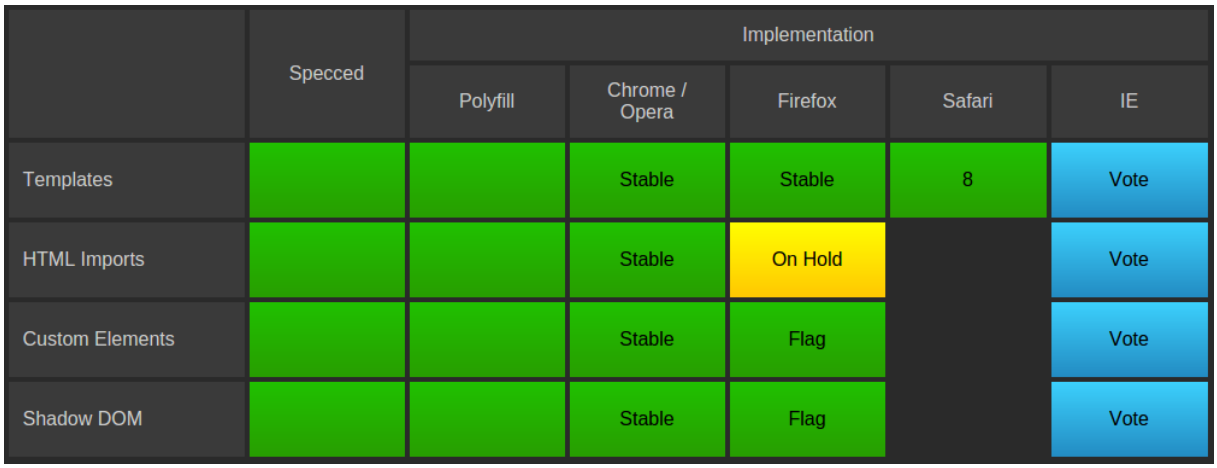


Figure 5.7: Browser support for the four Web Component specifications.[2]

```
1 <html>
2   <head>
3     <link rel="import" href="my-element.html">
4   </head>
5   <body>
6     <my-element></my-element>
7   </body>
8 </html>
```

Listing 5.13: Simple *index.html* importing *my-element.html* and using the custom element defined in there.

The Listing 5.13 shows, how the *my-element.html* is imported on line 3 and how the element `<my-element>` is used on line 6 like any other HTML element.

Web Components give us reusable custom components, style and HTML encapsulation, the ability to hide functionality behind a single HTML tag and everything is shippable in a single HTML file.

5.6 Browser Support

Browser support for the Web Component specifications varies from vendor to vendor. Chrome and Opera fully implement all four specifications, whereas Firefox has implemented most of them. Safari has only implemented the `<template>` element, probably because its part of the HTML5 specification. The status for the other three specifications is unknown. Microsoft is currently holding a vote whether to implement the specifications or not.

[2]Source: https://jonrimmer.github.io/are-we-componentized-yet (Retrieved: 06.07.2015)

5.7 Polymer

Polymer [3] is a library developed by Google to abstract the native Javascript APIs of the Web Component specifications. It also allows to define a custom element using just declarative HTML. Functionality is added by using Javascript and the life cycle callbacks of Polymer, that are similar to the callbacks of the Custom Elements specification described above.

```
1  <polymer-element name="awall-avatarcard" attributes="user">
2    <template>
3      <style>...</style>
4      <img id="img" class="avatar" src="{{user.avatar}}" alt="{{user.name}}" title
           ="{{user.name}}">
5    </template>
6    <script>
7      Polymer(Polymer.mixin({
8        ready: function () {
9          var data = this.user;
10         this.touch.makeDraggable(this.$.img, { clone: true, data: data });
11       }
12     }, window.appMixin));
13   </script>
14 </polymer-element>
```

Listing 5.14: Definition of a custom element with Polymer.

Listing 5.14 shows how the custom element `<awall-avatarcard>` is defined using Polymer. A custom element defined with Polymer uses the three relevant specifications, namely Shadow DOM, Custom Elements and Templates. Apart from that, is also offers data-binding, as seen in line 4 using double mustaches. In the example, the data comes form the attribute `user` that is defined on line 1. This means, that the `<awall-avatarcard>` element should be given the object containg the user's data as an attribute with `<awall-avatarcard user="someUserObj">`.

A `<template>` element and a `<script>` element are enclosed in the Polymer element definition. In the `<template>` element, the declarative part of the element is defined using HTML. In the `<script>` tag, the functionality of the element is implemented in the `Polymer()` function call, as seen in lines 6 to 13. The code uses the `ready` callback to make the `<img>` element on line 4 draggable. The `<img>` element is referenced with `this.$.img` on line 10, whereby the `img` part is the ID of the HTML element on line 4.

[3] https://www.polymer-project.org/0.5/

Chapter 6

Implementation

6.1 Frameworks

The *aWall* web application uses Polymer [3], a library that abstracts Web Components and allows to define custom elements using declarative HTML tags instead of pure Javascript code. This makes it easy to write new custom elements without the huge amount of boilerplate code that is required when using the native Javascript APIs directly. For multi-touch gestures like pinch-to-zoom, drag & drop and resizability of elements, *aWall* uses the interact.js [1] library.

6.2 Core Modules

The core of the application is implemented in *core.js*. The file contains three modules (Figure 6.1) defined using the self-executing function pattern for encapsulation. The modules are accessible through the global `window` object in the namespace app.* (e.g. app.core or window.app.core). The module `app.core` contains everything for loading and managing workspaces. `app.touch` offers functions to make HTML elements interactable by providing functions to activate drag & drop and resizability. The `app.util` module offers some utility functions used by the other two.

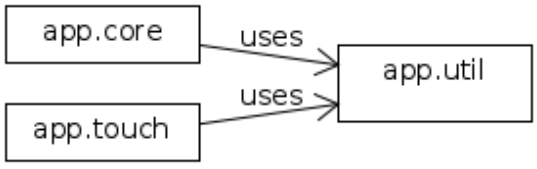


Figure 6.1: Overview of the modules defined in core.js.

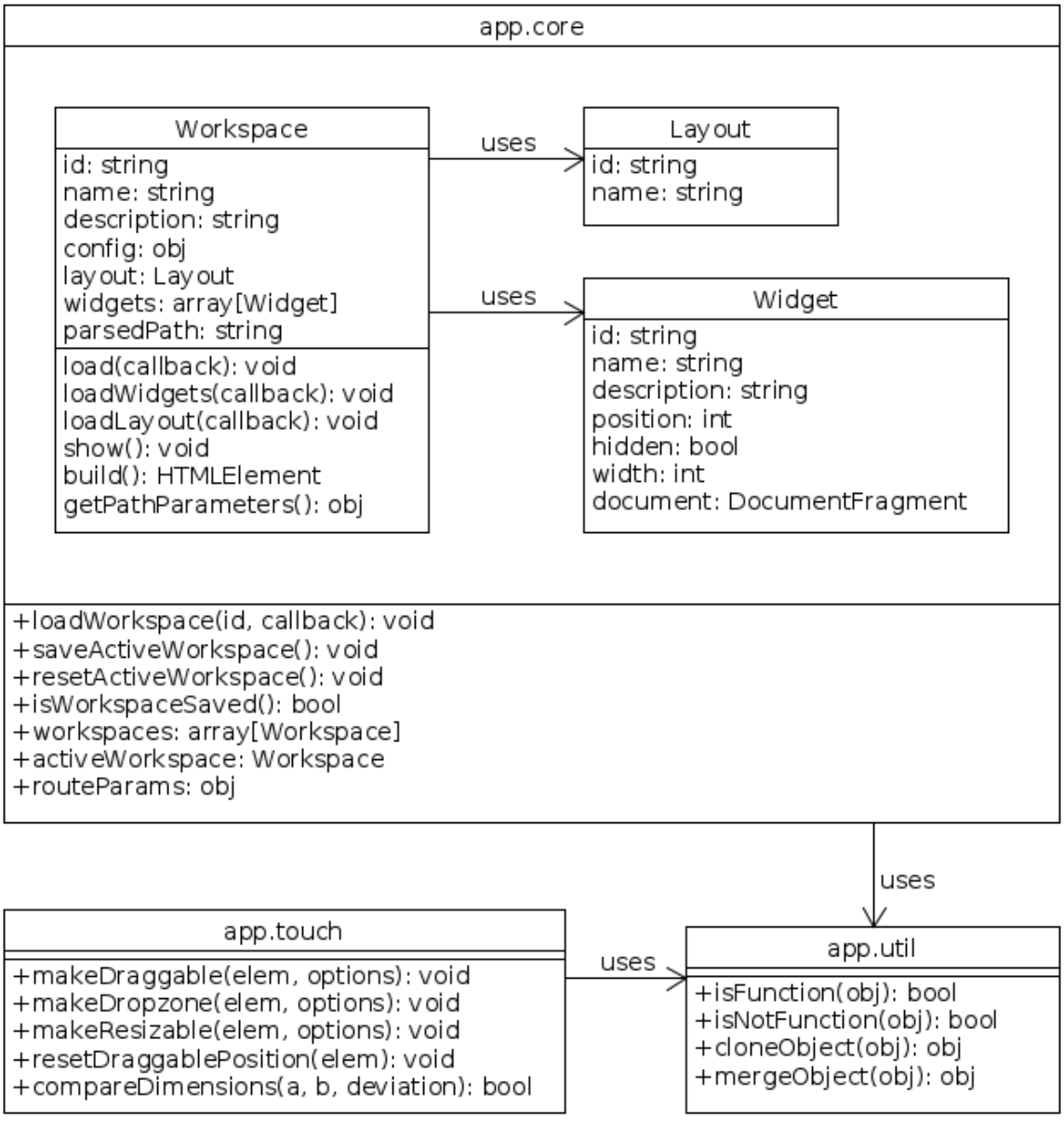


Figure 6.2: The modules defined in *core.js* and their public interface.

6.2.1 Asynchronous Loading

Javascript code runs in a single-threaded virtual machine in the browser. Loading a file from a networked resource takes a long time in computer terms and thus synchronous loading would waste a lot of time. Javascript is designed to be asynchronous and loading multiple resources asynchronously at the same time makes the language much more responsive, but it also means that the pattern of calling a function and expecting the return value to be the result of the asynchronous operation does not work with Javscript. Instead, a callback function is supplied that is called after the asynchronous operation has been completed. This creates a different flow compared to structured or object-oriented languages like C or Java.

```
1 // synchronous call
2 var result = load(someValue);
3
4 // asynchronous call
5 load(someValue, function(result) {
6   // use result
7 });
```

Listing 6.1: Synchronous vs. asynchronous call of a function.

To load, for example, multiple widgets with one call using a function `loadWidgets()`. It needs to keep two variables: The number of widgets to load and the number of widgets that have been loaded. The function to load a single widget is asynchronous too and the `loadWidgets()` supplies a callback function (called once the widget has been loaded) that increments the number of widgets loaded. At the same time, the code in this callback function checks if the number of loaded widgets equals the number of widgets to load. If that is the case, the callback function supplied to `loadWidgets()` is called, indicating that all widgets have been loaded.

6.3 Application Start-up

The web application loads all workspace configuration files first, then determines which workspace to load by examining the fragment identifier of the Uniform Resource Identifier (URI). The requested workspace is then loaded by importing the HTML file that defines the layout and all the configured widget's HTML files. Some custom elements are used in multiple widgets and are thus defined in their own HTML document. For example the custom element that represents a task, called `<awall-taskcard>`, is defined in *elements/awall-taskcard.html*. Those elements are imported directly by using the HTML Imports specification in the layout or the widget that uses it. Once the layout and the widgets have been loaded, the widgets are instantiated and inserted into the layout

according to their configuration attributes like position and width. The layout is then set as an element in the body of the DOM-tree.

```
1  var workspaces = [
2    "projectselect",
3    "sprintselect",
4    "workspaceselect",
5    "sprintplanning2",
6    "taskboard"
7  ];
8  window.addEventListener("load", function () {
9      workspaces.forEach(function (workspaceId) {
10       app.core.loadWorkspace(workspaceId);
11     });
12 })
```

Listing 6.2: Application starting point in *index.html*.

The Javascript code in Listing 6.2 shows the code bootstrapping the application in *index.html*. The only thing needed to load all the workspace configurations is the ID, which represents the name of the JSON configuration file without the file ending (e.g. sprintplanning2 -> *sprintplanning2.json*). The configurations are loaded on lines 9 to 11 in a callback that is called as soon as the page has been loaded on line 8.

6.4 Workspaces

A workspace is represented by a class with the same name (Figure 6.2) in the module `app.core`. The class has all the properties of the configuration. It keeps the configuration object as a property and the layout and the widgets are object references. The `parsedPath` property (Figure 6.2) is a computed property. It returns the fragment identifier of the workspace derived from the configuration's `path` property with all the variables filled with values.

The Javascript code in Listing 6.2 calls `loadWorkspace()` on line 10 which is shown in Listing 6.3 from line 1 to 9 as pseudocode. The configuration is loaded from LocalStorage [1] if available, otherwise from the remote web server. The configuration is checked against some constraints like required fields and the requirement for exactly one main widget. If that checks out, a `Workspace` object is created and added to the module's list of workspaces on line 7. On line 8, the function `routeChanged()` is called, which checks the fragement identifier in `Workspace.getPathParameters()` against the workspace configuration's `path` property. If it returns a non-null value, this workspace becomes the active workspace. `routeChanged()` is registered as a global callback for `"hashchange"` with the code in Listing 6.4 and is called every time the fragment identifier

[1] https://developer.mozilla.org/en-US/docs/Web/API/Storage/LocalStorage

of the URI changes. The object returned by `Workspace.getPathParameters()` contains the variables of the configuration's `path` as keys and the values are the values from the fragment identifier (e.g. the path `"/:pId/sprints/:sId"` results in `{"pId": 1002, "sId": 345}`). It is then globally available through the module's `routeParams` object which is set on line 14. Finally, the `Workspace.show()` function is called that loads, builds and shows the workspace.

`Workspace.show()` loads the layout and the widgets defined in the configuration. If either of them have already been loaded, the functions on line 20 and 21 return immediately. On line 22 to 25, the current workspace's layout element is queried by ID and removed. Then the requested workspace is built with `Workspace.build()` on line 27 (function definition starts on line 31). The workspace is built by instantiating the layout and adding all the widgets to it. To load and instantiate a widget, the `<template class="content">` element is queried, that is defined in the widget's *index.html* file, on line 37 and on line 38 the template is cloned using `document.importNode()` with the second argument indicating to clone all the descendants of the imported node (deep). Then, the widget's `view` property is set to the imported node (the widget's view) and the `widget` object is added to the layout on line 39. After all widgets have been created and added to the layout, the layout is returned on line 40 to the variable `rootNode` on line 27. It is the layout element on which the ID `"workspace"` is set and finally, it is added to the document's body on line 29.

```
loadWorkspace(id, callback):
  var config = loadFromLoacalStorage(id);
  if config === null:
    config = loadFromServer(id);
  check(config);
  var workspace = new Workspace(id, config);
  workspaces.add(id, workspace);
  routeChanged();
  callback(workspace);

routeChanged():
  foreach ws in workspaces:
    if ws.getPathParameters() !== null:
      routeParams = ws.getPathParameters();
      activeWorkspace = ws;
      activeWorkspace.show();
      break;

Workspace.show():
  this.loadLayout();
  this.loadWidgets();
  // remove current active workspace
  var activeRootNode = document.getElementById("workspace");
  if activeRootNode !== null:
    document.body.removeChild(activeRootNode);
  // build and add new workspace
  var rootNode = this.build();
  rootNode.setAttribute("id", "workspace");
  document.body.appendChild(rootNode);

Workspace.build():
  // create layout element
  var layout = document.createElement("layout-" + this.layout.id);
  this.layout.document = layout;
  // add widgets to layout element
  foreach widget in this.widgets:
    var template = widget.document.querySelectorAll("template.content")[0];
    widget.view = document.importNode(template.content, true);
    layout.addWidget(widget);
  return layout;
```

Listing 6.3: Simplified pseudocode without asynchronous calls to load, build and show a workspace. Emphasized keywords are module variables of *app.core* (Figure 6.2).

```
// Add global #-change listener
window.addEventListener("hashchange", routeChanged);
```

Listing 6.4: Registration of the `"hashchange"` listener that gets called when the fragment identifier changes. Pseudocode for `routeChanged()` is shown in Listing 6.3

6.5 Layouts

All the layouts, currently `layout-single` and `layout-workspace`, in the framework have to extend the `app-layout` custom element (Figure 6.3). The name of a layout's custom element must have the format layout-*, where * is the name of the layout, for instance *workspace* which results in `layout-workspace`. The HTML file containing a layout's custom element definition must be named after the name of the layout (e.g. `layout-workspace` is defined in *layouts/workspace.html*).

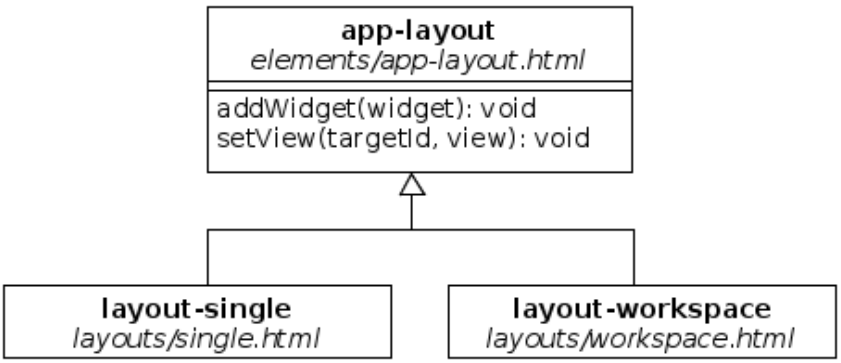


Figure 6.3: The two layouts of *aWall* with their base element `<app-layout>`.

```
Workspace.loadLayout():
  if this.layout === undefined:
    var id = this.config.layout;
    loadHtml("layouts/" + id + ".html");
    this.layout = new Layout(id);

loadHtml(url):
  var link = document.createElement("link");
  link.rel = "import";
  link.href = url;
  document.head.appendChild(link);
```

Listing 6.5: Pseudocode that shows how a layout is loaded.

The pseudocode in Listing 6.5 shows how a layout is loaded. If the layout variable is not set yet, the HTML file containg the layout's custom element definition is loaded

using the aforementioned conditions about its name. The `loadHtml()` function imports the HTML file using the HTML Imports specification by creating the `<link>` element with the corresponding attributes with the constructed URL and then appends it to the document's head on line 11. Finally the `layout` property of the workspace is set in line 5.

`app-layout`, the base element of layouts has two functions: The first is `addWidget()` that is used by external code when loading the workspace. It adds a widget to the layout. By default, it calls the second function `setView()` as follows: `this.setView("widget" + widget.position, widget.view)`. The `setView()` function queries for the element with the ID given as first argument `targetId`. Then creates a shadow root for encapsulation and adds the second argument `view` to that shadow root. Given the line called by `addWidget()`, the `<template>` element of the layout must contain elements with IDs in the form of widget*, where * is `"main"` for the main widget or an integer for the additional widgets (e.g. `<div id="widgetmain" />` or `<div id="widget1" />`).

The *aWall* application currently has two layouts:

- `layout-single`: Only contains a `<div>` element with the ID `"widgetmain"`.
- `layout-workspace`: Is the full workspace with a main widget, the info-view with additional widgets and a workspace-menu. It overwrites the `addWidget()` function to create dropzones for the info-view widgets.

6.6 Responsive Widgets

```
1  Workspace.loadWidgets():
2    if this.widgets.length <= 0:
3      foreach id,config in this.config.widgets:
4        var wCfg = loadJson("widgets/" + id + "/widget.json");
5        var widget = new Widget(id);
6        widget.document = loadHtml("widgets/" + id + "/index.html");
7        widget.name = wCfg.name;
8        widget.description = wCfg.description;
9        widget.position = config.position;
10       widget.width = config.width;
11       widget.hidden = config.hidden;
12       this.widgets.push(widget);
```

Listing 6.6: Pseudocode that shows how the widgets of a workspace are loaded. The `loadHtml()` function is defined in Listing 6.5.

All the widgets use the custom element `<awall-widget>`. It gives them the features like drag & drop and resizability. The four views of the widgets are also implemented

by custom elements, namely `<awall-widget-title>`, `<awall-widget-size-default>` , `<awall-widget-size-full>` and `<awall-widget-editable>`. Listing 6.7 shows the pattern on how a widget is implemented using the aforementioned elements. The title for a widget is given as the parameter to the attribute `value` for the element `<awall-widget-title>`. For the other three view elements, the content is more comprehensive than just a string and thus is enclosed in the element's tags as indicated using HTML comment syntax.

```
1  <awall-widget>
2    <awall-widget-title value="Team" />
3
4    <awall-widget-size-default>
5      <!-- Content for default view -->
6    </awall-widget-size-default>
7
8    <awall-widget-size-full showHeight="500" showWidth="400">
9      <!-- Content for full view -->
10   </awall-widget-size-full>
11
12   <awall-widget-editable showHeight="500" showWidth="400">
13     <!-- Content for edit mode -->
14   </awall-widget-editable>
15 </awall-widget>
```

Listing 6.7: HTML elements for the widget's different views.

Not all the view elements must be used of course. A widget does not need to have a full view or an edit mode. Both are optional and can be configured using the two parameters `showHeight` and `showWidth` to define their breakpoints at which point they activate and the default view is hidden. The edit mode is not shown automatically, but is represented by an edit icon appearing when available as depicted in Figure 3.3d.

The `<awall-widget>` custom element implements the features that all widgets have and controls when which view is displayed. Listing 6.8 shows part of the template of the `<awall-widget>` element. The `<content>` element is used in Shadow DOM and represents an insertion point for DOM-trees. That means that everything between the start and end tag of `<awall-widget-size-default>` in Listing 6.7 is inserted in the position of the content tag that selects the `<awall-widget-size-default>` in line 4 of Listing 6.8.

```
<content select="awall-widget-title" />

<div id="extendedContent">
  <content select="awall-widget-size-default" />
  <content select="awall-widget-size-full" />
  <content select="awall-widget-editable" />
</div>
```

Listing 6.8: Part of the `<awall-widget>`'s template using insertion points for the different HTML tags.

All elements in the template in Listing 6.8 except `<awall-widget-title>` are enclosed in a `<div>` element whose visibility is controlled by the media query shown in Listing 6.9. As long as the browser's viewport height is smaller than the defined `max-height` breakpoint, only the title of the widget is shown. When the widget is dragged out of the info-view, the default view becomes visible. This is not controlled by a media query but by Javascript code since the height of the viewport has not changed. The widget can now be resized using the pinch-to-zoom gesture or the mouse. During resizing, the width and height of the widget are compared to the `showHeight` and `showWidth` breakpoints defined for the full view and the edit mode.

```
@media only screen and (max-height: 40rem) {
  #extendedContent {
    display: none;
  }
}
```

Listing 6.9: Media query hiding all the views except the title when the screen is small enough.

6.7 Sprint Planning 2 Widget

As mentioned before, each widget is responsible for its own responsive behavior. The main widgets use the `<awall-widget>` but with drag & drop and resizability disabled. The main widget for the SP2 workspace, as shown in Listing 6.10, has two templates that are controlled by the boolean variable `isSmallScreenHeight`. The variable is set by the application's Javascript code and is accessible through Polymer's data binding mechanism. Depending on the value of the variable one of the two inactive templates is activated. The views are all compartmentalized into their own custom elements and thus the two templates in Listing 6.10 only contain one element (results in Figure 3.4a) for the small-screen view and two elements for the large-screen view, that results in Figure 3.2. The `<awall-userstory-cardlist>` element for small screens handles its behavior to switch to the detailed view (Figure 3.4b) itself. The title for the widget at the top of the listing is visible on any screen and thus is not enclosed in a `<template>` element.

```
<h2>Sprintplanning 2</h2>

<template if="{{isSmallScreenHeight}}">
  <awall-userstory-cardlist />
</template>

<template if="{{!isSmallScreenHeight}}">
  <awall-sprintbacklog />
  <awall-temptasks />
</template>
```

Listing 6.10: How the SP2 main widget decides which view to show.

6.8 Communication

6.8.1 Server-side

The server-side implementation of the REST API was developed in the previous ADS project using C# and Microsoft's ASP.NET Web API. The controllers in Figure 6.4 represent the top layer of the application server. All concrete controllers representing a real URL (e.g. `TasksController`) inherit from the base class `AbstractApiController` which implements HTTP methods like `GET` and `POST`. It also uses the `AwallController`, the WebSocket endpoint, in the methods `Post()`, `Put()` and `Delete()` by calling the `Notify()` method which broadcasts the change notifications to all connected clients.

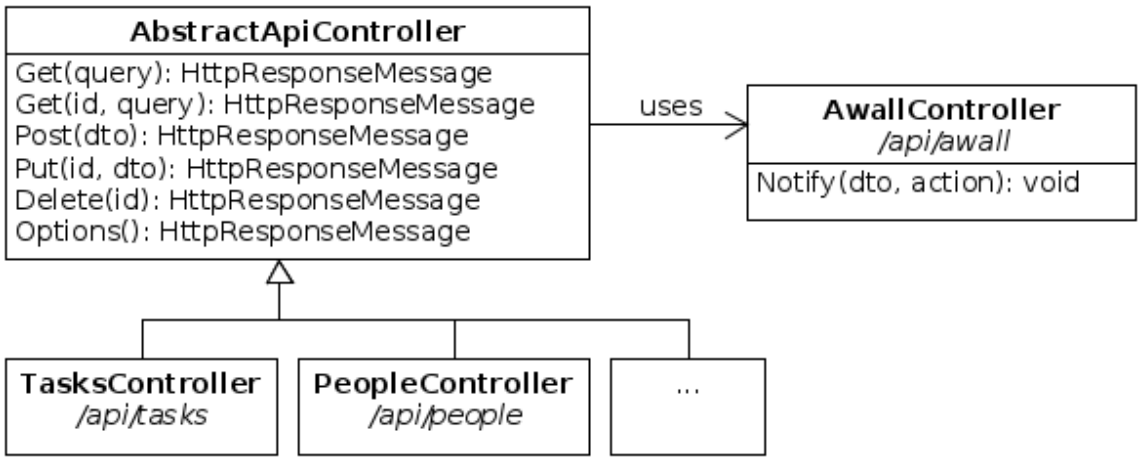


Figure 6.4: How the controllers of the ADS REST API use the WebSocket endpoint.

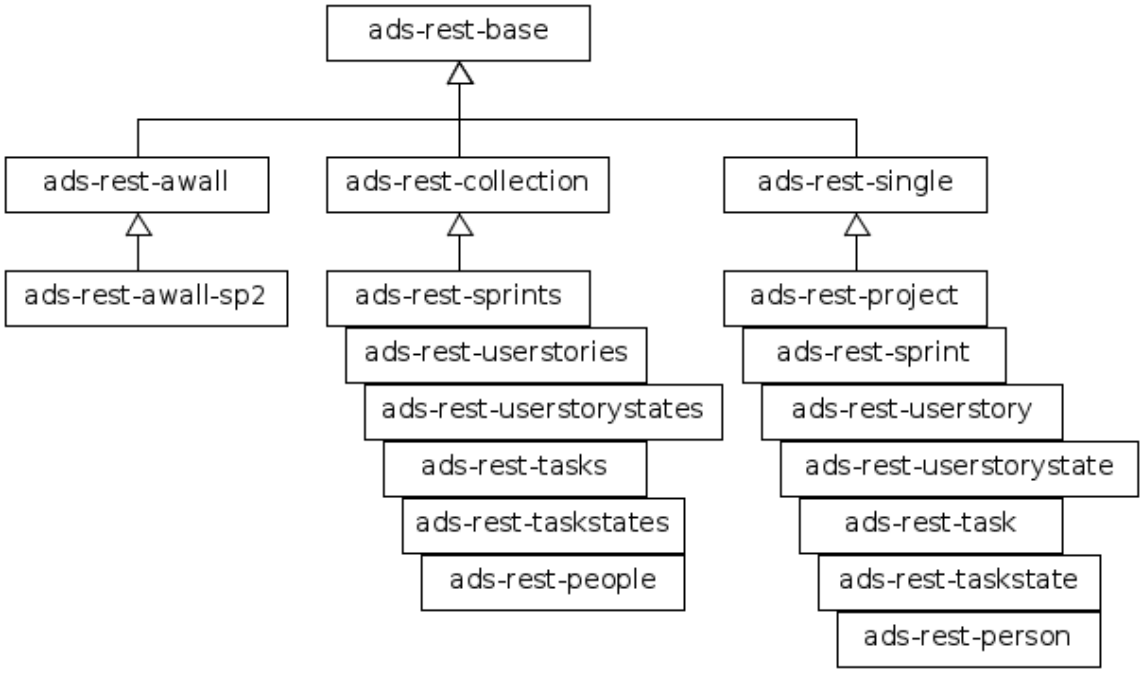


Figure 6.5: Class diagram of all ads-rest-* elements.

6.8.2 Client side

The ADS REST API and the WebSocket endpoint are accessible through custom elements (Figure 6.5) that have the prefix ads-rest-* and are defined in *elements/ads-rest.html*. For each resource of the REST API there are two elements: One for the collection resource (e.g. `ads-rest-tasks` for */api/tasks*) and one for the single resource URL (e.g. `ads-rest-task` for */api/tasks/{id}*).

Figure 6.6 shows an extended class diagram including functions using the resource type `Task` as an example. All elements inherit from `ads-rest-base` which provides the connection to the WebSocket endpoint in a singleton (self-executing function) and functions to register observers with `registerObserverFor()` based on `resourceType` (e.g. `ads-rest-collection` registers an observer with the `resourceType` of the concrete sub-class like `ads-rest-tasks` with `resourceType: "Task"`). `registerObserverFor()` in turn registers itself with the WebSocket code, that calls the `onDataChanged()` function on the registered elements when a message with the corresponding resource type has been received. The `onDataChanged()` function implementation in `ads-rest-base` checks the messages and then calls the function corresponding to the `action` property of the message: `onUpdated()`, `onDeleted()` and `onCreated()`. Those functions are all implemented by the base class for collection resources `ads-rest-collection` and only the function `onUpdated()` for single resources. To send messages to the server, `ads-rest-base` provides the function `send()`.

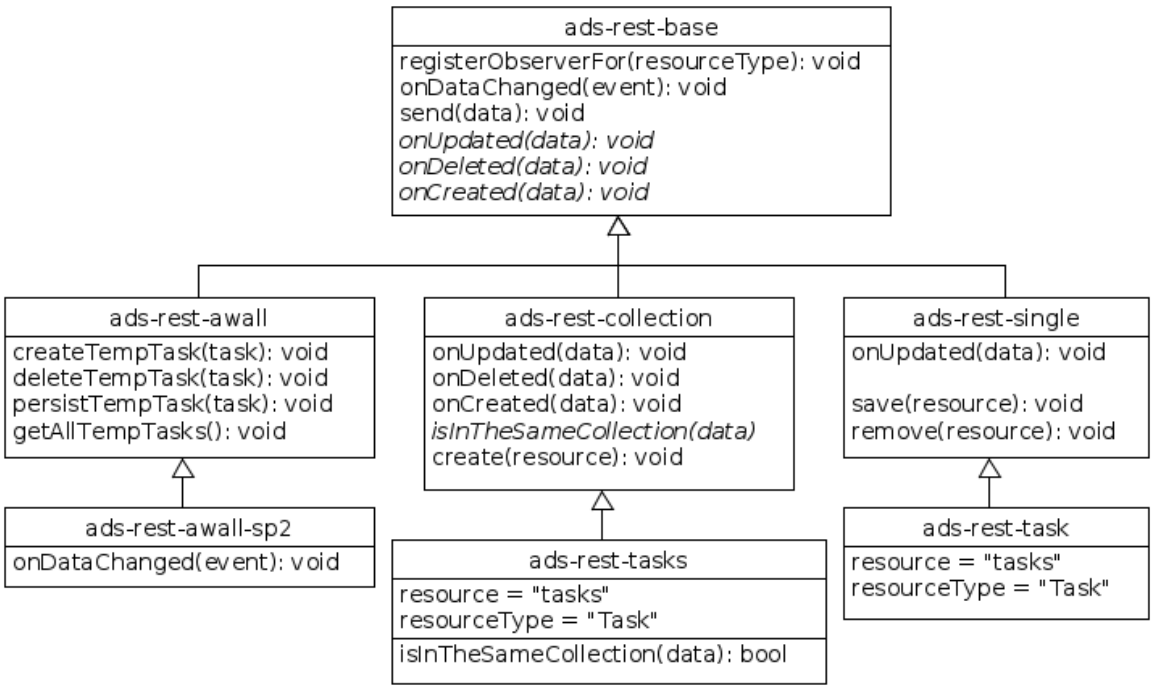


Figure 6.6: Class diagram of ads-rest-* elements including important functions.

`ads-rest-awall` provides some convenience functions to send pre-defined messages to the server like the function `persistTempTask(task)` that sends a message with the action `PersistTemp`, resource type `Task` and as data the given task. The element `ads-rest-awall-sp2` inherits from `ads-rest-awall` and overwrites the `onDataChanged()` function of `ads-rest-base` to handle the messages more specifically. The element provides a list of the unassigned tasks to the user of the element, that is updated in-place when a change occurs unsing Polymer's data-binding.

The `ads-rest-collection` element is the base class for all REST API resources that represent a collection (e.g. */api/tasks*) and to create a new resource with the function `create()`, that issues a HTTP `POST` request behind the scenes. It implements the three `onXXXX` function of `ads-rest-base` and updates the array with the result (e.g. a list of tasks) of the request accordingly. To do that, the sub-elements must implement the function `isInTheSameCollection()` which ensures that only a list containing data relevant to the change notification is updated. The element `ads-rest-tasks` is an element to be used and inherits from `ads-rest-collection`. It implements the `isInTheSameCollection()` function and provides the name of the resource for the URL with the property `resource` and the resource type for the WebSocket communication with `resourceType`.

The element `ads-rest-single` is the base element for all elements representing a single instance of a resource like a single task. It only implements the `onUpdated()`

function, because the other two are not relevant for the single instance. Apart from that, the element provides the function `save()` to save the modified resource (issues a HTTP `PUT` request to the server) and the function `remove()` to delete the instance on the server (issues a HTTP `DELETE` request). As an example of a single instance element, the `ads-rest-task` is shown in Figure 6.6. The element provides the name of the resource for the URL with the property `resource` and the resource type for the WebSocket communication with `resourceType`.

6.9 Interactions

The module `app.touch` in core.js (Figure 6.2) provides a simple API for touch interactions. The API abstracts the interact.js [1] library. The following list shows which function of `app.touch` which interact.js function calls:

- Dropzone (`makeDropzone()`): `interact.dropzone()`
- Draggable (`makeDraggable()`): `interact.draggable()`
- Resizable (`makeResizable()`): `interact.resizable()` for mouse interaction, `interact.gesturable()` for pinch-to-zoom on a multi-touch screen.

6.9.1 Drag & Drop

Drag & drop consists of two parts: The draggable element that can be moved around and a dropzone into which the draggable can be dropped into.

Dropzone

A dropzone is a normal HTML element that has been made into a dropzone by using the function `makeDropzone()` in the module `app.touch` as shown in Figure 6.2. The first parameter is the HTML element that should be made into a dropzone and the second parameter is an options object. The following options can be used to configure and interact with a dropzone:

- `accept`: The CSS selector that the draggable must have in order to be accepted by this dropzone (visualized in Figure 6.7).
- `data`: The `data` attribute in the object supplied to the `onenter` and `onleave` callback functions.
- `ondrop`: Function called when a draggable has been dropped in this dropzone. Has one object as parameter with the following attributes:

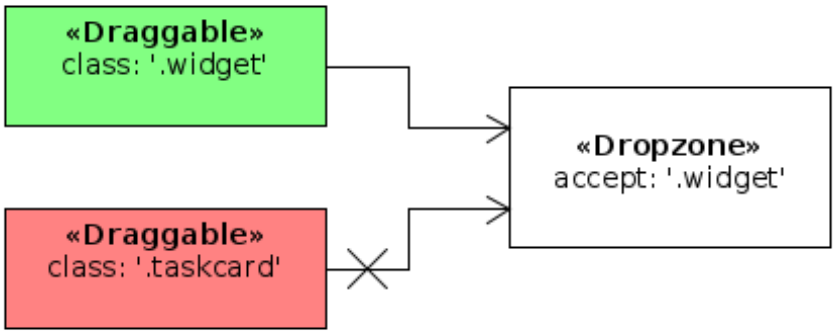


Figure 6.7: How a dropzone accepts draggables.

  - `draggableData`: The `data` attribute supplied to the draggable as `data` when it was created.
  - `draggableElement`: The root node of the draggable element.
- `onenter`: Function called when a draggable has entered the dropzone. Has one object as parameter with the following attributes:
  - `data`: The `data` attribute supplied when the dropzone was created.
  - `draggableElement`: The root node of the draggable element.
- `onleave`: Function called when a draggable has left the dropzone. Has one object as parameter with the following attributes:
  - `data`: The `data` attribute supplied when the dropzone was created.
  - `draggableElement`: The root node of the draggable element.

The dropzone offers some CSS classes to indicate what is happening. This allows the developer to change the look of the dropzone, for example highlighting it when a draggable is moved to indicate that it accepts that draggable. The available CSS classes are as follows:

- `drop-active`: A draggable accepted by this dropzone is being dragged.
- `drop-entered`: A draggable acepted by this dropzone has entered the dropzone and can be dropped.

Draggable

Every element in the DOM-tree can be made into a draggable with the `makeDraggable()` function in the module `app.touch` (Figure 6.2). The first parameter to the function is

the element to make into a draggable. The second argument is an options object to configure and interact with the draggable and has the following attributes:

- `clone`: Boolean value indicating whether the draggable should be cloned while being dragged. The draggable element stays in place and the clone is visibly dragged.
- `data`: The `draggableData` attribute supplied to the dropzone's `ondrop` function.
- `revert`: Boolean indicating whether the draggable should revert back to its starting point when it has not been dropped in a dropzone.
- `absolute`: Boolean indictaing whether the draggable should use absolute values for moving. Is only necessary when the draggable element changes its parent element (e.g. is removed from a node and added to another DOM element).
- `onstart`: Function called when the draggable starts to move.
- `onmove`: Function called repeatedly when the draggable is moved.
- `onend`: Function called when the draggable has stopped moving.
- `ondrop`: Function called when the draggable has been dropped into a dropzone.

Like the dropzone, the draggable has CSS classes so the developer can configure the look of a draggable with the classes as follows:

- `can-drop`: The draggable has entered a dropzone and can be dropped.
- `dragging`: The draggable is being dragged.

6.9.2 Resize

The `app.touch` module also allows to make an element resizable using the mouse by dragging the corner or using the pinch-to-zoom gesture on a multi-touch screen. The function `makeResizable()` (Figure 6.2) has two parameters: The first is the element that should be made resizable, the second parameter is an options object to register callbacks. The possible callbacks are:

- `onstart`: Called when resizing has been started.
- `onmove`: Called repeatedly when resizing.
- `onstop`: Called when resizing has stopped.
- `disableMove`: Allows to disable resizing by returning true. Is used to disable resizability when an info-view widget is docked.

Chapter 7

Discussion

7.1 Hardware

The setup for the multi-touch wall consists of 2x2 monitors. In between the monitors there is a small border around each monitor as seen in Figure 2.1. Interaction across these borders is possible, but they are a haptical obstacle that is not always successfully crossed.

Because of the 2x2 monitor setup, the touches are detected by an additional bezel around the system using infra-red. The system has some difficulties detecting two distinct touch-points when the distance between two fingers is too narrow. This is especially noticeable when using the pinch-to-zoom gesture.

Both of these problems can be avoided by using a single high-resolution wall display that uses a capacitive touch-screen instead of infra-red.

7.2 UI Design

A nice feature of tablets is that the user can hold it however he wants. The orientation of the screen changes with how the user holds the device, which is either portrait or landscape mode. This confronts the designer or developer with another variation of screen resolution that is challenging because the two modes have different aspect ratios. That often means that the UI for portrait mode is not adequate for the landscape mode. *aWall* is optimized only for landscape mode on all devices.

On multi-touch devices without a physical keyboard a virtual on-screen keyboard is provided by the system. The keyboard is not integrated in the same way on all systems. For example on a Microsoft Surface tablet running Widows 8.1, the virtual keyboard overlays everything and the content is not aware of it and thus can not show the relevant content, e.g. a form, in the small space left above the keyboard. On the other hand, the virtual keyboard in Android is much more integrated into the operating system and thus

the content in an application is automatically moved to the part of the screen not hidden by the keyboard.

A convenient option for text input on a tablet is using a pen instead of the virtual keyboard. There are not many tablets on the market that support pen input. One product line that does are the Microsoft Surface Pro tablets. The support is built into the virtual keyboard application and handwriting detection works pretty well. The problem with the keyboard hiding the application's content persists with the pen input since it is the same application handling the input.

7.3 Responsive Design

In combination with new best practices like RWD, enabled by the new standards, the development of new and adaptable web applications becomes more approachable. Especially media queries allow us to detect and react to changes like the screen resolution that are important for the rendering of content. They allow us to either hide elements or render them using a different style (e.g. smaller font). Together with the HTML `<template>` element, this allows us to create inactive DOM-trees in conditional templates that can be shown or hidden depending on a boolean variable set by a media query.

The main widget for the SP2 workspace presented in this thesis uses such conditional templates to present its content depending on the size of the browsers viewport dimensions. The views including their specific behavior for both display sizes I implemented, namely the large wall and the tablet, have to be developed separately. There is no silver bullet that generates all the views automatically. Thanks to the use of custom elements, more complex elements can reuse other custom elements.

7.4 Implementation

The architecture I developed for *aWall* shows, that current web technologies and emerging new web standards make it possible to build an application that serves different devices with disparate screen resolutions and physical size using a single code base.

Through its design of using loosely-coupled Web Components, specifically custom elements with Shadow DOM, there are no dependencies between the widgets. This allows a developer to create new widgets without worrying about whether there are going to be problems with other widgets like giving an element an ID that has already been used or having its stylesheet definitions (CSS) affect other elements of other widgets.

By working with the newest technologies, there is always the problem with rapidly evolving and changing APIs and libraries. Since I started to develop *aWall*, the Polymer library evolved and reached version 1.0 with lots of changes. It is going to take some time to port the current application developed with Polymer 0.5 to the new version.

Another inconvenience of using the newest web technologies is that the number of browsers implementing all those new features is limited. Currently, *aWall* only runs natively on the Chrome and Opera web browsers. Most other browsers are in progress of implementing the specifications for Web Components. For all those browsers not supporting the features, there are so called polyfills available. Polyfills are Javascript libraries, that add the missing APIs to the browser. One big disadvantage of using polyfills is their lower performance compared to the native implementations.

The application is a SPA and loads all its resources dynamically. Information about the sprints, user stories and tasks is fetched using a request/response REST API. Notifications about create, update and delete operations are propagated to all connected clients over a WebSocket connection. For example, when someone on another device modifies a task, all running *aWall* applications receive a notification about the change and the data is replaced in-place, without requiring a page reload or any other interaction of the user. This ensures that the displayed data on all running instances of the *aWall* web application is always up-to-date.

The *aWall* web application consists of lots of HTML files because it makes frequent use of the new HTML Imports specification. This may impact the load time for the web application because browsers open only about 6-8 connections simultaneously and each file requires a new connection. There is a way to create single HTML file containing all Javascript, CSS and HTML by using a tool called vulcanize [1] that has been developed specifically to minimize web applications written with Polymer. I tried multiple times to minimize my application with it and failed every time. I could not determine the exact reason for the failure, but i suspect it is the structure of my application using dynamic loading of resources (in *core.js*) instead of using the HTML Imports specification everywhere. Another possible solution to this problem would be to use the new HTTP/2 [5] protocol for the web server, which uses just one TCP connection to fetch all files using multiple asynchronous streams.

7.5 Validation

The requirements were defined at the beginning of the project and are listed in the appendix in chapter B. The following sections have the same structure as the requirements in the appendix and explain how the requirements are implemented in the *aWall* web application.

[1] https://github.com/polymer/vulcanize

7.5.1 Component-based Web Application

Software Architecture The components of the system, namely widgets, layouts and elements, are all independent. The workspace configuration combines the layout with the widgets to form a workspace. Dependencies, like elements (components) providing access to the REST API, are referenced by using their custom HTML tag directly by the components needing them. All widgets, layouts and elements may depend on elements that are globally available. Services are defined in elements and components with a view are the widgets. Components with a view (widgets) do not offer functionality respectively services to others except for view interactions like drag & drop to interchange some specific information. The elements are defined as custom HTML elements and just need to be inserted by using their HTML tag by another element to be used (dependency). The whole web application is hosted on a web server, where the layouts, widgets and elements are defined in their own directories and files. Adding new components to the system is relatively easy, as chapter A shows.

Communication between Components Communnication between the components per se like a publish-subscribe system is not available, but Polymer's bi-directional data-binding allows the application, for instance, to update the list of tasks in-place in all components that use this list. That way, advanced communications infrastructure inside the application is not needed.

View Layouts All widgets may have up to four views: The title-only view (requirements: tiny), the default view (requirements: card), the full view and an edit mode. Only the smaller widgets in the info-view have those views. They are controlled by the layout, whereas the main widgets are themselves responsible to adjust their view to the size of the viewport.

Gestures and Multi-touch All the widgets in the info-view can be moved around freely and can be resized using the mouse or the pinch-to-zoom gesture on a multi-touch display.

View Interaction Since the requirements were defined, the concept of the main working area has changed. The main working area is called the main widget and is developed specifically for a meeting of the agile process (e.g. the main widget for the SP2 meeting). For each meeting of the agile process a workspace is defined, which is a combination of a layout, the meeting's main widget and multiple additional widgets for the info-view. The main widget is fixed and can not be replaced by dropping another widget into its

place. Instead the user can click on a link in the navigation bar at the bottom to switch to another meeting's workspace.

For the requirement about information interchange, the *aWall* application implements this for several interactions: The team view widget displays a list where all team members are represented by an avatar. The avatar picture can be dragged to a task in any of the workspaces to assign this person to this task. Another example is the SP2 main widget, where unassigned tasks can be dragged to a user story and dropped, whereby the task is assigned to this user story. Or in the taskboard workspace, the tasks can be moved from from one column to another, for example from 'Todo' to 'In Progress', to change the task's state.

Collaborative Aspect Each instance is connected to the WebSocket endpoint of the application server. Each change (create, update, delete) made with any instance of the application is propagated to all connected clients (web application instances) in real-time. So, if a task created on a tablet, it immediately pops up on the wall and on any other device running the *aWall* web application.

7.5.2 Responsive Design

Layouts The main widgets are responsible to adjust their content to the size of the viewport themselves. In the current implementation, the SP2's main widget reacts to the height of the viewport to switch between a layout for large screens like the wall and a layout with two views for smaller screens like a tablet. The split up layout displayed on a tablet is divided into an overview, showing only the user stories of a sprint, and a detail view for a user story. There, the selected user story is displayed with all its assigned tasks and a panel for the unassigned tasks.

To minimize the waste of precious space on smaller screens, the layout uses the title-only view (requirements: tiny) for the widgets in the info-view. When dragging them out of the info-view, they expand to their default view and are usable like on the large screen.

Supported Devices The *aWall* application is optimized fo the multi-touch wall with a 4K resolution and 10" tablets with a 1080p resolution, both in landscape mode. The application looks good on a desktop/laptop display too. Smartphones are not supported, respectively would require a different layout and different interactions as on the larger screens.

Device Type Detection The device type is not detected but rather the size of the browser's viewport using media queries (RWD).

Chapter 8

Summary & Outlook

In this thesis, I designed and implemented a software architecture for a web application based on loosely-coupled components with a responsive user interface. The loose coupling and encapsulation for the individual components is achieved by using Web Components, a set of four emerging web technologies. The responsive UI is built around media queries with which the application can adapt itself to the size of the browser's viewport and offer an optimal viewing experience on the large wall as well as on the tablet. The application is based on configurable workspaces, consisting of a layout, a main widget and multiple smaller widgets that can be moved around freely and resized.

Most of requirements defined at the beginning of the project have been met, whereas some have been implemented exactly as defined because the technology used in the implementation was not known to me back then. Additionally, the layout behavior was changed during development. Nonetheless, the web application I developed works as intended and is easy to extend in the future with new layouts, widgets and elements.

The next steps in the development could be:

- Upgrade the Polymer library to the most recent version [1].
- Implement more widgets like a timer for the daily stand-up or a burn-down chart.
- Create the workspaces for the other meetings of the agile process.
- Make use of the full view and the edit mode for info-view widgets.
- Implement the default/full and edit mode for task and user story cards.
- Implement features to conduct the meetings with a distributed team (e.g. visualize the movement of a drag & drop operation on all instances of the application or implement a widget for audio/video chat using WebRTC [6])

[1] Polymer 1.0 at the time of writing

Appendix A

How to add ...

A.1 A Workspace

Create a new JSON configuration file in the directory *workspaces* (the format is shown in chapter 4.3). Then add the name without the .json ending to the `workspaces` array in the application's *index.html*.

A.2 A Widget

Create a new directory with the name of the widget (e.g. teamview). Then create a *widget.json* (Listing A.1) and an *index.html* (Listing A.2) file in that directory. The additional views for the widget like full view and edit mode are described in chapter 6.6. To use the newly created widget, add it as a widget in the desired workspace configuration with the name of the directory as key in the `widgets` object (described in chapter 4.3).

```
{
  "name": "Team",
  "description": "Shows the members of the project."
}
```

Listing A.1: Example of a *widget.json* file.

```
<template class="content">
  <awall-widget>
    <awall-widget-title value="Dummy widget 1"></awall-widget-title>
    <awall-widget-size-default>
      <p>Lorem ipsum dolor ...</p>
    </awall-widget-size-default>
  </awall-widget>
</template>
```

Listing A.2: Simple example of an *index.html* file without the full view and edit mode.

A.3 A Layout

Create a HTML file with the name of the layout in the *layouts* directory. For instance *single.html*. The following requirements must be met in order for the layout to work (example in Listing A.3):

- The name of the polymer element the must coincide with `layout-*`, where * is the name of the HTML file **.html* (e.g. *single.html* -> `layout-single`).
- The Polymer element must extend `app-layout`.
- The template must at least have a container like a `<div>` element for the main widget with the ID `"widgetmain"`. Additional widgets are numbered, starting at 1 (the `position` property in the workspace configuration) with IDs like `"widget1"` (Add a container with this ID: `<div id="widget1"></div>`).

To use the layout, add it as `layout` property in a workspace configuration (described in chapter 4.3).

```
<link rel="import" href="../elements/app-layout.html">

<polymer-element name="layout-single" extends="app-layout">
  <template>
    <style>
      <!-- style definitions -->
    </style>
    <div id="widgetmain"></div>
  </template>
  <script>
    Polymer({});
  </script>
</polymer-element>
```

Listing A.3: Example of a simple layout (*single.html*).

Appendix B

Requirements

The following requirements were defined at the beginning of the project and were written down in the project declaration (German: Projekterklärung) [26].

B.1 Component-based Web Application

Software Architecture

1. The components of the system should be independent, but there is a framework where the components register themselves and announce their services and dependencies.
2. There are two types of components: Components without a view who offer services like access to the REST API and Components with a view. The latter may also offer services to other components.
3. All the components are located on the same web server (same domain), but may have their own directory.
4. The development of new components should be easy.
5. The components should be reusable by other components.

Communication between Components

1. The framework offers facilities for the components to communicate with each other. This includes changes to the underlying data. For example, if there are two components using the same data and a user changes the data with component A, component B should be informed of the changes.

View Layouts

1. Components with a view may have up to three views:
   - **Full**: The full view shows all the available information and is shown in the main working area.
   - **Card**: A smaller view where only the important information is displayed. Is shown around the main working area.
   - **Tiny**: Shows only the title and the icon of the component.

Gestures and Multi-touch

1. The view components can be moved around with drag & drop and be resized using the mouse or touch (resize with pinch-to-zoom).

View Interaction

1. The view components can be dragged around in the card view and dropped into the main working area, where it expands to the full view.
2. Is the main working area already occupied, the current component minimizes to the card view and moves to the border.
3. The system should be able to interchange information between components with drag & drop gestures. For example: Component A contains a list of avatars representing the team. The user should now be able to assign a team member by dragging the avatar of component A to an element in component B to assign this person.

Collaborative Aspect

1. A team should be able to conduct the sprint planning 2 meeting by using multiple devices to simultaneously create new tasks for a user story on tablets.
2. The created tasks should immediately become visible on the wall after being created on the tablet and displayed in a separate panel.
3. There, they can be edited, deleted or assigned to a user story.

B.2 Responsive Design

Layouts

1. The components have different view layouts depending on the size of the display.

2. Some components may not be able to offer all functionality on all device types in the same way.

Supported Devices

1. The application may offer support for up to four display sizes (4K on the wall, desktop and 10” tablet with 1080p and smartphones with a 1080p/720p resolution).
2. The main focus are the large multi-touch wall and the 10” tablet. The other device types are nice to have.

Device Type Detection

1. The device type should be detected to select the correct layout, if possible/necessary.

Appendix C

Tools

C.1 Hardware

Development machine

Type	Lenovo x230 ThinkPad
OS	Fedora Linux and Windows 8.1
Screen resolution	1366 x 768 on 12.5" monitor 1920 x 1200 on 24" monitor with docking station
Touchscreen	No
Pen	No

Tablet

Type	Microsoft Surface 2 Pro
OS	Windows 8.1
Screen resolution	1920 x 1080 (1080p) on 10.6" monitor
Touchscreen	Yes (capacitive)
Pen	Yes

Multi-touch Wall

Type	PQLabs Multi-Touch Wall
OS	Windows 8.1
Screen resolution	3840 x 2160 (4K) on 2x2 46" monitors
Touchscreen	Yes (infra-red)
Pen	No

C.2 Software

Javascript and HTML

WebStorm 9.x / 10.x [1]
NPM (Node package manager) [2]
Bower package manager [3]
Chrome

C#

Visual Studio 2013 Ultimate

Server

Windows 2012 R2 Enterprise
IIS 8.5 with .NET Framework 4.5.x

[1] https://www.jetbrains.com/webstorm
[2] https://www.npmjs.com
[3] http://bower.io

Bibliography

[1] Taye Adeyemi. 2015. interact.js. http://interactjs.io. (2015). Accessed: 2015-05-30.

[2] Atlassian. 2015. JIRA – The flexible and scalable issue tracker for software teams. https://www.atlassian.com/software/jira. (2015). Accessed: 2015-06-16.

[3] Polymer Authors. 2015. Polymer. https://www.polymer-project.org. (2015). Accessed: 2015-05-30.

[4] Sriram Karthik Badam and Niklas Elmqvist. 2014. PolyChrome: A Cross-Device Framework for Collaborative Web Visualization. In *Proceedings of the Ninth ACM International Conference on Interactive Tabletops and Surfaces (ITS '14)*. ACM, New York, NY, USA, 109–118. DOI:http://dx.doi.org/10.1145/2669485.2669518

[5] M. Belshe, R. Peon, and Ed. M. Thomson. 2015. Hypertext Transfer Protocol Version 2 (HTTP/2). https://tools.ietf.org/html/rfc7540. (2015).

[6] Adam Bergkvist, Daniel C. Burnett, Cullen Jennings, and Anant Narayanan. 2015. WebRTC 1.0: Real-time Communication Between Browsers. http://www.w3.org/TR/webrtc/. (2015).

[7] Apoorve Chokshi, Teddy Seyed, Francisco Marinho Rodrigues, and Frank Maurer. 2014. ePlan Multi-Surface: A Multi-Surface Environment for Emergency Response Planning Exercises. In *Proceedings of the Ninth ACM International Conference on Interactive Tabletops and Surfaces (ITS '14)*. ACM, New York, NY, USA, 219–228. DOI:http://dx.doi.org/10.1145/2669485.2669520

[8] Alistair Cockburn. 2001. *Agile Software Development*. Addison-Wesley Professional.

[9] WebComponents.org contributors. 2015. WebComponents.org – a place to discuss and evolve web component best-practices. http://webcomponents.org. (2015). Accessed: 2015-05-27.

[10] I. Fette and A. Melnikov. 2011. The WebSocket Protocol. https://tools.ietf.org/html/rfc6455. (December 2011). Accessed: 2015-05-27.

[11] Y. Ghanam, Xin Wang, and F. Maurer. 2008. Utilizing Digital Tabletops in Collocated Agile Planning Meetings. In *Agile, 2008. AGILE '08. Conference*. 51–62. DOI: http://dx.doi.org/10.1109/Agile.2008.13

[12] Dimitri Glazkov. 2014. Custom Elements – W3C Working Draft. http://www.w3.org/TR/custom-elements. (16 December 2014). Accessed: 2015-06-17.

[13] Dimitri Glazkov and Hayato Ito. 2014. Shadow DOM – W3C Working Draft. http://www.w3.org/TR/shadow-dom. (17 June 2014). Accessed: 2015-06-17.

[14] Dimitri Glazkov and Hajime Morrita. 2014. HTML Imports – W3C Working Draft. http://www.w3.org/TR/html-imports. (11 March 2014). Accessed: 2015-06-17.

[15] Stevenson Gossage. 2014. Understanding the Digital Card Wall for Agile Software Development. (May 2014). Master thesis.

[16] Benedikt Haas and Florian Echtler. 2014. Task Assignment and Visualization on Tabletops and Smartphones. In *Proceedings of the Ninth ACM International Conference on Interactive Tabletops and Surfaces (ITS '14)*. ACM, New York, NY, USA, 311–316. DOI:http://dx.doi.org/10.1145/2669485.2669538

[17] Ian Hickson, Robin Berjon, Steve Faulkner, Travis Leithead, Erika Doyle Navara, Edward O'Connor, and Silvia Pfeiffer. 2014. HTML5 – W3C Recommendation. http://www.w3.org/TR/html5/scripting-1.html#the-template-element. (28 October 2014). Accessed: 2015-06-17.

[18] Martin Kropp and Andreas Meier. 2015. Swiss Agile Study 2014. http://www.swissagilestudy.ch/files/2015/05/SwissAgileStudy2014.pdf. (May 2015). Accessed: 2015-05-29.

[19] Ethan Marcotte. 2011. *Responsive Web Design*. A Book Apart.

[20] Magdalena Mateescu, Martin Kropp, Roger Burkhard, and Carmen Zahn. 2015. aWall – Designing Socio-Cognitive Tools for agile team collaboration with large Multi-Touch Wall Systems. (October 2015). To be submitted.

[21] Sumit Pandey. 2013. Responsive Design for Transaction Banking - a Responsible Approach. In *Proceedings of the 11th Asia Pacific Conference on Computer Human Interaction (APCHI '13)*. ACM, New York, NY, USA, 291–295. DOI:http://dx.doi.org/10.1145/2525194.2525271

[22] Florian Rivoal, Håkon Wium Lie, Tantek Çelik, Daniel Glazman, and Anne van Kesteren. 2012. Media Queries – W3C Recommendation. http://www.w3.org/TR/css3-mediaqueries. (19 June 2012). Accessed: 2015-05-27.

[23] Jessica Rubart. 2014. A Cooperative Multitouch Scrum Task Board for Synchronous Face-to-Face Collaboration. In *Proceedings of the Ninth ACM International Conference on Interactive Tabletops and Surfaces (ITS '14)*. ACM, New York, NY, USA, 387–392. DOI:http://dx.doi.org/10.1145/2669485.2669551

[24] Stefan Röthlisberger. 2014a. Agile Data Services. (7 February 2014).

[25] Stefan Röthlisberger. 2014b. Agile Multi-Touch Task Board. (30 June 2014).

[26] Stefan Röthlisberger. 2014c. Project Declaration for the Master Thesis 'Component-Architecture for Loosely-Coupled Responsive Webapplications'. (3 October 2014).

[27] Andrew Stellman and Jennnifer Greene. 2014. *Learning Agile*. O'Reilly Media.

[28] Ryan Sukale, Olesia Koval, and Stephen Voida. 2014. The Proxemic Web: Designing for Proxemic Interactions with Responsive Web Design. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication (UbiComp '14 Adjunct)*. ACM, New York, NY, USA, 171–174. DOI:http://dx.doi.org/10.1145/2638728.2638768

[29] Luke Wroblewski. 2011. *Mobile First*. A Book Apart.

List of Figures

List of Listings

List of Tables