APLIKAČNÍ RÁMEC PRO VÝVOJ INFORMAČNÍCH SYSTÉMŮ V JAZYCE DART

WEB APPLICATION FRAMEWORK FOR SOFTWARE DEVELOPMENT IN THE DART LANGUAGE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE MIROSLAV RAŠKA
AUTHOR

VEDOUCÍ PRÁCE RNDr. MAREK RYCHLÝ, Ph.D.
SUPERVISOR

BRNO 2014

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2013/2014

Zadání bakalářské práce

Řešitel: **Raška Miroslav**
Obor: Informační technologie
Téma: **Aplikační rámec pro vývoj informačních systémů v jazyce Dart**
Application Framework for Information System Development in Dart
Kategorie: Informační systémy

Pokyny:

1. Seznamte se s jazykem Dart, jeho specifikací a aplikačním rozhraním dostupných knihoven.
2. Seznamte se s obecnými možnostmi použití a problematikou tvorby softwarových rámců. Poveďte analýzu a srovnání vlastností několika softwarových rámců pro implementaci aplikací v jazycích podobných jazyku Dart.
3. Navrhněte vlastní softwarový rámec pro podpru implementace aplikací v jazyce Dart. Soustřeďte se zejména na architekturu aplikací a aplikační komponenty a možnosti jejich integrace a spolupráce, které popište pomocí vhodných architektonických a návrhových vozrů.
4. Možnosti softwarového rámce ilustrujte na návrhu ukázkové aplikace v jazyce Dart.
5. Po konzultaci s vedoucím implementujte navržený softwarový rámec jako knihovnu jazyka Dart. Implementujte také ukázkovou aplikaci využívající softwarový rámec.
6. Zhodnoťte dosažené výsledky a navrhněte možná rozšíření.

Literatura:

- Ralph E. Johnson. *Documenting Frameworks as Patterns*. In Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '92), Vancouver, Canada, 1992. ISBN 0-201-53372-3. [http://dx.doi.org/10.1145/141936.141943]
- Ademar Aguiar, Gabriel David. *Patterns for documenting frameworks: customization*. In Proceedings of the 2006 conference on Pattern languages of programs (PLoP '06). ACM, New York, USA, 10 pp., 2006. ISBN 978-1-60558-372-3. [http://doi.acm.org/10.1145/1415472.1415491]
- Leesa Murray, David Carrington, Paul Strooper. *An approach to specifying software frameworks*. In Proceedings of the 27th Australasian conference on Computer science - Volume 26 (ACSC '04), Australian Computer Society, Darlinghurst, Australia, p. 185-192, 2004. ISBN 1-920682-05-8. [http://crpit.com/confpapers/CRPITV26Murray.pdf]
- Dart API Reference. [http://api.dartlang.org/docs/releases/latest/]
- The Dart Programming Language Specification (0.20, M2). The Dart Team. [http://www.dartlang.org/docs/spec/latest/dart-language-specification.html]

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 4.

Vedoucí: **Rychlý Marek, RNDr., Ph.D.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2013
Datum odevzdání: 21. května 2014

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

[illegible]

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Brno University of Technology - Faculty of Information Technology

Department of Information Systems Academic year 2013/2014

Bachelor Project Specification

For:	**Raška Miroslav**
Branch of study:	Information Technology
Title:	**Web Application Framework for Software Development in the Dart Language**
Category:	Information Systems

Instructions for project work:

1. Make yourself familiar with the Dart language, its specification and available libraries.
2. Analyse the concepts, applications, and potential of software frameworks and the issues related to the software frameworks design. Make a comparison of features of several software frameworks for the Dart language and similar, focus particularly on web application frameworks.
3. Propose a novel web application framework supporting software development in the Dart language. Design its architecture and architecture of its applications, utilised components, details of their communication and cooperation, and suitable architectural and design patterns.
4. Demonstrate possibilities of application of the framework, propose a sample application.
5. After agreement with the supervisor, implement the proposed application framework as a Dart language library. Implement also the sample application based on the framework.
6. Evaluate the results and discuss further extensions and future work.

Basic references:

- Ralph E. Johnson. *Documenting Frameworks as Patterns*. In Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '92), Vancouver, Canada, 1992. ISBN 0-201-53372-3. [http://dx.doi.org/10.1145/141936.141943]
- Ademar Aguiar, Gabriel David. *Patterns for documenting frameworks: customization*. In Proceedings of the 2006 conference on Pattern languages of programs (PLoP '06). ACM, New York, USA, 10 pp., 2006. ISBN 978-1-60558-372-3. [http://doi.acm.org/10.1145/1415472.1415491]
- Leesa Murray, David Carrington, Paul Strooper. *An approach to specifying software frameworks*. In Proceedings of the 27th Australasian conference on Computer science - Volume 26 (ACSC '04), Australian Computer Society, Darlinghurst, Australia, p. 185-192, 2004. ISBN 1-920682-05-8. [http://crpit.com/confpapers/CRPITV26Murray.pdf]
- Dart API Reference. [http://api.dartlang.org/docs/releases/latest/]
- The Dart Programming Language Specification (0.20, M2). The Dart Team. [http://www.dartlang.org/docs/spec/latest/dart-language-specification.html]

Detailed formal specifications can be found at http://www.fit.vutbr.cz/info/szz/

Supervisor:	**Rychlý Marek, RNDr., Ph.D.**, DIFS FIT BUT
Beginning of work:	November 1, 2013
Date of delivery:	May 21, 2014

L.S.

Dušan Kolář
Associate Professor and Head of Department

Abstrakt

Vývoj webových aplikacích se potýká se specifickými problémy, které by mohly být vyřešeny novým webovým aplikačním rámcem kombinujícím moderní technologie a nový přístup k návrhu aplikací. Jednotlivé problémy webových aplikací jsou nastíněny včetně stavu jejich řešení v současných webových rámcích. Představena je architektura řízená zprávami, kostra aplikačních komponent a rozličná rozšíření. Popsány jsou problémy při implementaci obecného řešení v jazyce Dart. Dopady jednotlivých rozhodnutí a řešení problémů jsou ilustrovány na ukázkách reálných webových aplikací.

Abstract

Web applications development nowadays is experiencing specific difficulties in presentation layer that could be solved by a framework that combines modern technologies and novel framework approach. The difficulties are introduced, along with their solution in existing web frameworks. Novel, message-driven framework architecture, basic component structure and various framework extensions are analysed and outcomes and effects are discussed. Technical problems with implementation of generally analysed solutions in the Dart language are examined. The decisions and solutions are accompanied by their effects on real-world applications.

Klíčová slova

Webový aplikační rámec, webový rámec, rámec v jazyce JavaScript, rámec v jazyce Dart, softwarová architektura, Cloud aplikace, Internetová aplikace, architektura řízená událostmi, architektura orientovaná na služby, modulární architektura, škálovatelná architektura, rozšiřitelný rámec, modulární rámec, škálovatelný rámec, informační systémy, Dart, JavaScript.

Keywords

Web application framework, Web framework, JavaScript framework, Dart framework, Software architecture, Cloud application, Rich Internet application, RIA, Single-page application, Event-driven architecture, Service-oriented architecture, SOA, Modular architecture, Scalable architecture, Extensible framework, Modular framework, Scalable framework, Information systems, Front-end framework, Client-side framework, Dart, JavaScript.

Citace

Miroslav Raška: Web Application Framework for Software Development in the Dart Language, bakalářská práce, Brno, FIT VUT v Brně, 2014

Rozšířený abstrakt

Vývoj webových aplikacích se potýká se specifickými problémy, které by mohly být vyřešeny novým webovým aplikačním rámcem kombinujícím moderní technologie a nový přístup k návrhu aplikací. Tato práce popisuje průběh návrhu tohoto rámce ve třech logických celcích: úvod do oblasti tvorby webových aplikací, analýza obecného řešení včetně architektury rámce a průběh implementace rámce v jazyce Dart.

Úvod přibližuje architekturu moderních webových aplikací, především pak těch interaktivních. Dále představuje původní motivaci pro vytvoření nového aplikačního rámce a hlavní cíle. Např. že by rámec měl být univerzální, jednoduchý k použití a předcházet častým chybám při vývoji Internetových aplikací. Zároveň by měl být použitelný v komerční sféře, proto jsou zanalyzována kritéria, na základě kterých firmy vybírají webový rámec pro své produkty. Následně jsou představeny tři ukázkové aplikace, na kterých budou v průběhu práce ilustrovány dopady či výhody zvolených řešení.

Analýza se nejprve zabýva současnými webovými rámci a zkoumá, zda se podobné řešení již nevyskytuje, případně v jakém rozsahu. Také zkoumá slabá místa současných rámců a knihoven, tedy možná témata, kterými by se měl nový rámec zabývat, aby byl přínosný. Poté je popsán průběh návrhu základní architektury, od modulární architektury, přes škálovatelnou architekturu až po architekturu s centrální sběrnicí. Tato architektura kombinuje výhody všech předchozích architektur a je zároveň kompatibilní s architekturami řízenými událostmi a architekturami orientovanými na služby. Jsou také definována základní pravidla pro bezpečnou komunikaci na sběrnici.

V druhé části analýzy je definována základní kostra aplikačních komponent a zabýva se především přehledností kódu vytvářených komponent a jejich konfigurací. Dále pak vnitřní bezpečností rámce, tak aby byla vynucena pravidla pro tvorbu komponent a aby tak komponenty vždy spňovaly určité základní vlastnosti. Na základě těchto zaručených vlastností je poté představeno několik zajímavých rozšíření, která jsou obvykle implementována velmi složitě, avšak vlastnosti komponent a architektura rámce umožňuje jejich elegantní implementaci. Tato rozšíření jsou např. získání stavů komponent a jejich obnova a sdílení, univerzálně využitelné grafické komponenty, logická hierarchie komponent, testování uživatelských interakcí či pokročilá obsluha výjimek. Také jsou představeny běžně používáné modely pro práci s daty.

V techické sekci je nejprve představen implementační jazyk Dart, jeho výhody a srovnání s ostatními klientskými webovými jazyky, včetně kompatibility s prohlížeči a rychlosti aplikací. Technická analýza se poté zabývá problémy vyplývajícími z vlastností jazyka Dart. Ten v některých případech neumožňuje, případně omezuje implementaci obecných řešení. Jsou tak postupně probrány všechny problémy, které bylo nutné vyřešit pro zdárnou implementaci minimální verze rámce, např. obecná rozšiřitelnost rámce a komponent, bezpečnost na sběrnici, rozdělení komponent do více tříd a jejich vzájemné provázání či efektivní statická analýza zdrojového kódu. Také je představeno několik vylepšení, která mají zjednodušit tvorbu aplikací. Nakonec je zmíněno několik zajímavých, nedořešených technických problémů.

Závěrem je zhodnocen přínos rámce, výhody navrhnuté architektury a zvolené struktury komponent. Je představen stav implementace, její slabé stránky a možnosti vylepšení. Rámec byl zveřejněn jako open-source software a na závěr jsou představeny budoucí kroky tohoto projektu.

Web Application Framework for Software Development in the Dart Language

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana RNDr. Marka Rychlého, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

....
Miroslav Raška
May 20, 2014

Poděkování

Děkuji svému vedoucímu, RNDr. Marku Rychlému, Ph.D., za odborné konzultace v oblasti architektur informačních systémů, za pomoc při plnění formálních požadavků této bakalářské práce a s ní souvisejících povinností a především za mnoho věcných a zajímavých připomínek, poznámek a návrhů.

Contents

Chapter 1

Introduction

Modern web applications development is experiencing speficic difficulties that could be solved by a web application framework described in this thesis. The framework is client-side only, designed primarily for creation of single-page applications running fully in a web browser. It is designed around architecture inspired by server-side solutions and features abstract, non-visual components only. This approach is new for client-side web application frameworks that are mostly focused on visual components and related features.

Identified difficulties and issues are described in the chapter 2, including possible solutions, their advantages and disadvantages. The framework architecture and technical details about suggested solutions follow in the chapter 3. Possible usage scenarios and benefits of chosen solutions are continuously illustrated with sample application examples. Introduction into the field of web application frameworks follows.

1.1 Layers in a modern web application

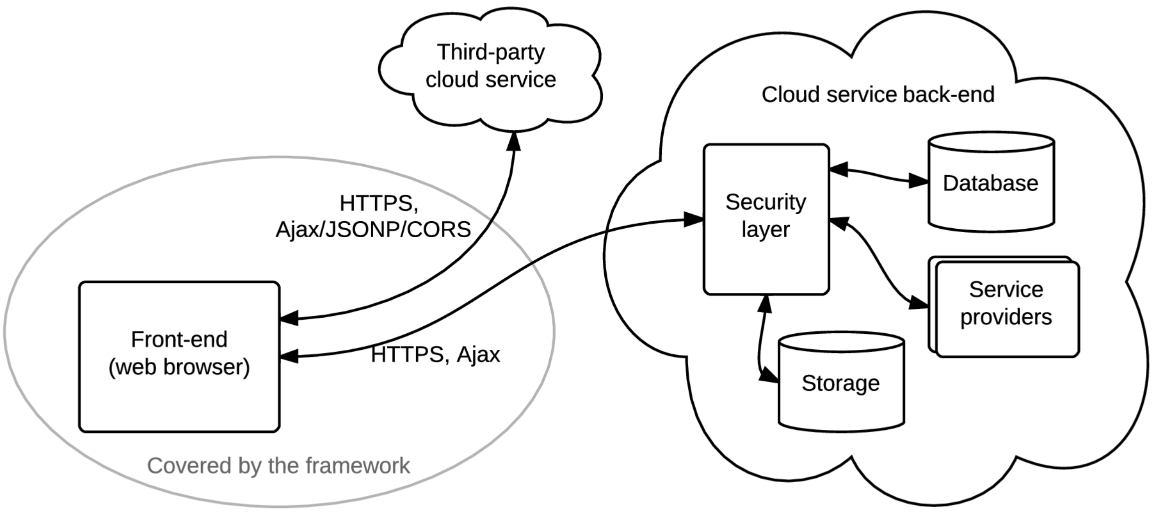


Figure 1.1: Architecture of a sample single-page cloud application.

Figure 1.1 shows simplified architecture of a sample interactive cloud application. Compared to a static application, the *front-end* contains most of the *business logic* code and all interactions handling code. *Back-end* serves only as an adaptor to the database and to a file storage. Because client-side can never be considered fully secure (e.g. HTTP requests can be sent independently of a web browser or sent using JavaScript console), back-end has to contain security layer that restricts user access to the database and other services. Additionally, the services that cannot be easily implemented within a web browser — like exports to PDF or time-consuming tasks — could be moved to the server-side.

Summed up, majority of the application-specific code is being moved to the front-end in many *Rich Internet applications* (RIAs) [22]. Within a *three-tier* architecture [21] the client part could include the whole *presentation layer* and most of the *domain layer*, leaving only the *data source layer* to the back-end.

1.2 Motivation

Not only technical tasks are being solved in this work, the original motivation of designing a framework was based on the following ideas where different business fields overlap, including marketing, project management or commercial insights. All of them are important during the software development process [32] and the software framework choice has influence on them.

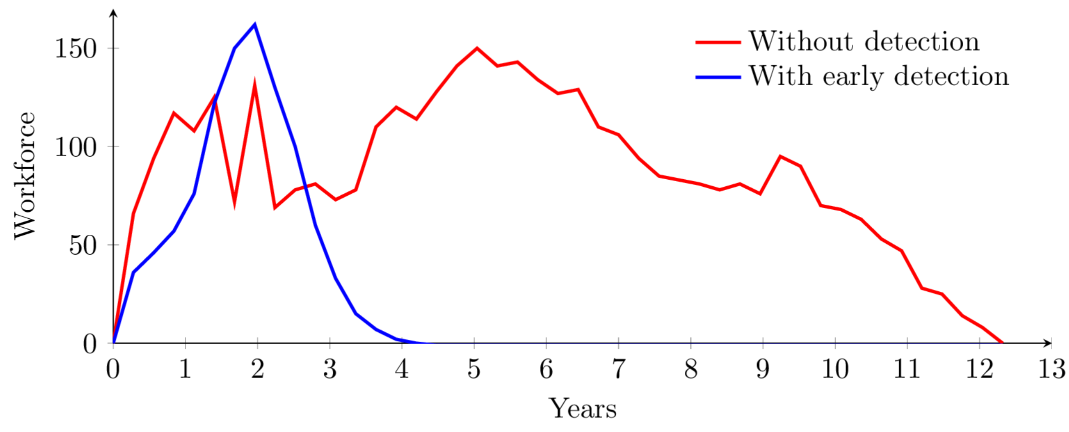
Goal 1: Design universal framework. This framework is focused primarily on applications with architecture similar to the one mentioned in the section 1.1. Particularly interesting are *single-page* information systems, enterprise applications and other rich Internet applications that bring interesting problems and issues to solve, e.g. scalability, internal application security or complex user interactions [32]. Yet the goal is to design universal framework that will be able to power any web application, not only enterprise-level ones. This is feasible because single-page information systems are complex applications and by supporting them any simpler application will be supported as well. Universal framework brings an opportunity to create broad community with insights from different areas of use, which could bring up many undiscovered issues or enhancements.

Goal 2: Make it easy to use. Application development is easier if the base framework is not complex and it is clear how to use it and where to start programming [25][39]. User should be able to develop applications after studying only few basic concepts, while following some basic recommendations and few simple rules. Framework should also reinforce writing clean code that is easily readable by others [30][39], because it increases productivity and eliminates project costs [30].

Goal 3: Prevent mistakes. Project duration and expenses can be decreased if mistakes and problems are detected early [24], as shown in the Figure 1.2, which compares actual projects with the same goal and objectives in the aeronautic industry. Project managed with problems detection mechanisms was finished three times faster and with significantly lower budged (in the figure signalized as an area below the curves). The results could be even better if some mistakes were prevented in advance, not just detected early. Framework should therefore contain methods to eliminate common code mistakes [39][32][30], meaning that specific constraints will be present and programmer will not be allowed to do absolutely anything or at least will be warned of possible drawbacks. Applications will be then more reliable and development process can be effectively managed [30].

Figure 1.2: Effects of early problems detection on the project duration and progress. [24]

.



Goal 4: Solve application-wide issues. Specialized issues can be solved by external libraries. Framework should primarily address the issues that are application-wide, e.g. different kinds of communication or interaction between components. Do not solve internal component issues that might be application specific or are already being solved in existing web frameworks or libraries. Do not deal with visual components, they were created many times and can be reused.

Goal 5: Consider business framework selection criteria. The Table 1.1 shows that most of the criteria used to select a framework or a library for the new company product do not refer to the technical features the framework provides. Important is also how would the choice influence future processes within the company [20][28]. The criteria should be exhaustive as they are based on different points of view — on a scientific research [20] and a business research company study [28].

The Table 1.1 also clearly shows which criteria are present in the most popular frameworks and libraries and which are contrarily overlooked by their creators. If the new framework addressed some uncovered issues, it would be a competitive advantage. Generally, these requirements should be considered when designing a framework in order to be competitive. A framework for developing business applications would be useless if not used by any company.

Goal 6: Align with the base technology The Dart language [6] was chosen as the base technology for this framework (details follow in the chapter 3). The framework should follow general principles and ideas behind the Dart language to be more familiar and easier to use to programmers already developing in the Dart language. The principal ideas extracted from the Dart issue tracker [11] and the official Dart website [33] are:

- **A language for mainstream.** "Dart was designed to look and feel familiar if you're coming from other languages." This implies that framework should be also designed to look familiar, e.g. should use standard constructions, naming conventions, standard classes and the same or similar API to the Dart libraries wherever possible.
- **Pragmatic approach to the performance.** The language features that might be useful but would have huge impact on virtual machine performance are not imple-

Table 1.1: Framework business selection criteria presence in selected web frameworks and libraries as of 2014-04-01 [20] [28] [10] [16] [9] [35].

	AngularJS	Backbone.js	Dojo	Ember.js	Ext.JS	jQuery + jQuery UI	Knockout	Meteor	MooTools	Prototype + script.aculo.us	Underscore.js	YUI	This framework[8]
Enterprise ready[1]	○	-	●	○	●	○	-	-	○	○	-	○	●
Broad functionality[2]	○	-	●	-	●	○	-	○	-	-	-	●	○
Business components[2]	○	-	●	-	●	○	-	○	-	-	-	○	○
Business logic support[3]	-	-	○	○	○	-	-	-	-	-	-	-	●
Clear deliverables	●	●	○	●	○	●	●	●	○	○	●	○	●
Consistent	●	●	-	●	○	○	●	●	-	○	●	○	●
Cross-browser	●	●	●	○	●	●	●	○	●	●	●	●	●
Easy to use	●	●	-	●	-	●	●	●	○	●	●	-	○
Extensible	●	●	●	●	●	●	●	○	●	●	-	●	●
Good documentation	●	●	●	●	●	●	●	-	●	●	●	●	●
Mature[4]	○	●	●	-	●	●	○	-	○	●	●	●	-
Other frameworks[5]	●	●	●	○	●	●	○	○	-	○	●	●	●
Scalable[6]	○	-	○	-	○	○	-	○	○	-	-	○	●
Software stability[7]	○	-	-	-	-	-	-	○	-	-	-	-	●
Updated regularly	●	●	●	●	●	●	●	●	-	-	○	●	●

Legend: - no support or information not available, ○ partial support, ● full support

[1] Based on official website claims; ○ mentioned support of highly interactive applications, ● enterprise usage explicitly mentioned.
[2] ● Supported directly, ○ supported partially, by community or 3rd-party extensions or libraries.
[3] Includes workflow and business processes support.
[4] Long time on the market, stable and without extensive API changes.
[5] Supports other simultaneously running instances of frameworks, does not modify native classes.
[6] Unused parts of the framework can be taken out, core components replaced with custom ones.
[7] Reinforces stability of the software created with the framework, as described in the section 1.2.
[8] Ideal state as of the first stable release. Some criteria are already satisfied by the Dart language.

mented. On the other side manual code performance optimizations do not make much sense, because compiler can take care of most of them, VM performance is improving with each release and browsers are more and more powerful over time.

- **User is smarter than analytic tools.** Enable freedom in what programmer does. Assume that user is smarter than any static analysis tool and code is written with some purpose. However, warn anytime something looks suspicious. Framework should therefore implement most constraints as warnings only, and should be extensible enough to enable replacement of any framework class or module if programmer decides to provide own solution.

Goal 7: Include recommendations Additionally to the API documentation the framework should contain set of recommendations and examples how to use it properly. These recommendations should also promote good programming practices, mention common mistakes and show examples of code misuse.

1.3 Sample applications

Following web applications were chosen to show the area of possible framework usage and to illustrate problems and solutions on real world examples. They are also partially implemented on the medium enclosed with this thesis.

App 1: Single-page content management system (CMS). Simple information system for updating contents of a blog or an article-based website. Containing 3 sections — articles, categories and users — with the basic *CRUD* functionality. It should demonstrate that framework is able to dynamically create and destroy individual components (views) and keep the *domain data* synchronized between views, e.g. when category is added it should be immediateliy available for selection in the view for creating new article.

App 2: Interactive email service. Web front-end to an email account, the basic functionality is simple CRUD on email domain class. However, additional interactive features can be present within single view, e.g. responsive filtering in the list, background task checking for new emails, folding of cited text, grouping of related messages or editable email drafts nested within email detail. Also global language or theme choice is performed dynamically without reloading the application.

App 3: Project management system. As an enterprise information system it requires additional features compared to previous B2C web services. Security policies are defined for each user separately, leading to different behaviour within views of the same entity class. For example user could have full access to the properties of own tasks, read-only access to colleagues' tasks and some properties might not be visible at all within boss's tasks. Access to the same entity can be simultaneous from more views and application should remain consistent even if this entity is removed remotely by other user during its lifetime. Additional features can be added by turning on application extensions or modules.

Chapter 2

Analysis

2.1 Analysis of current web frameworks

The Table 1.1 has already shown how existing frameworks meet business criteria. The next step is to compare selected technical features and architecture of these frameworks to find out whether it would be useful to create new framework and to determine which features to focus on.

The comparison summary is shown in the Table 2.1. The compared features were selected within 3 categories — architectural patterns, functionality and structure of visual components and non-visual components support. These are some areas that are related to the framework architecture [31]. The list of frameworks is based on the Table 1.1 and TodoMVC framework comparison site [3], with libraries and unmaintained frameworks excluded. Other 130 client-side frameworks and libraries were also taken into account but are not listed because they are either too specialized, unmaintained, incomplete or do not bring any innovative ideas or additional features when compared to the listed ones (for the complete list see Appendix B). Specialized mobile web application frameworks were also excluded. Table therefore displays mixture of popular, modern or innovative web frameworks.

2.1.1 Comparison results

AngularJS, Ember.js, Knockout and Meteor have similar UI concepts, they try to simplify creation of components by templates where model is automatically bound to view elements and is backwards updated with view changes. This functionality is called *two-way data binding* [37]. Dart language is equipped with Polymer package that is very similar and along with AngularJS supports modern HTML5 features and even feature drafts like custom elements [12] or shadow DOM [1]. Other frameworks use different approach where visual components are programatically defined from JavaScript. In some frameworks the views are not even separate classes or files, although the frameworks claim to support MV* architecture.

Generally, the frameworks are built around visual components, no regularly maintained abstract framework was present and almost no frameworks have good non-visually oriented architecture. The possible reasons are:

- Abstract frameworks are developed only as an experiment, a demonstration how frameworks could be done in a better way, but are not pushed into a stable or usable version.

Table 2.1: Comparison of technical features within selected web frameworks as of 2014-04-01.

	AngularJS	Backbone.js	Dojo	Ember.js	ExtJS	Knockout	Meteor	YUI	Dart + Polymer
Architectural patterns									
MV* architecture[1]	●	●	●	●	●	●	●	●	●
Event-driven architecture[2]	-	-	-	-	-	-	●	-	-
Service-oriented architecture[3]	○	-	-	-	-	-	-	-	-
Managed publish-subscribe[4]	●	-	●	-	○	-	●	-	○
Multi-level exceptions handling[5]	○	-	●	-	-	-	-	-	-
AMD compliant[6]	-	●	●	-	-	●	-	-	-
Visual components									
MV* within components	●	●	●	●	●	●	○	●	●
Separate HTML views	●	-	●	●	-	●	●	●	●
Automatic model-view binding[7]	●	-	-	●	-	●	●	-	●
Restricted component scope[8]	●	○	●	●	○	●	○	○	●
Native HTML5 views[9]	●	-	-	-	-	-	-	-	●
Encapsulated components[10]	-	-	-	-	-	-	-	-	●
Interaction testing support	●	-	●	●	-	-	-	-	-
Abstract components									
Unit-testing support	●	-	●	●	-	-	-	●	●
Internationalization	●	-	●	-	○	-	-	○	○
Component lifecycle mgmt.[11]	-	-	○	○	-	-	-	-	-

Legend: - no support or information not available, ○ partial support, ● full support

[1] Also called MVW (Model-View-Whatever); term covers e.g. MVC, MVVM and MVP design patterns [36][21][23].
[2] Event-driven architecture (EDA) is an architecture where any state change is broadcasted as event notification available to other application components [18].
[3] Service-oriented architecture (SOA) is an architecture where global functionality is encapsulated within standalone service providers. The components and providers are loosely-coupled through enterprise service bus (ESB) [34].
[4] Publish-subscribe (observer) design pattern [23] for handling events. Globally managed subscriptions prevent memory leaks and execution of code on destroyed objects by automatically removing all handlers when subscriber is destroyed. Explained in details in the section 2.2.3.
[5] Exceptions could be handled separately within specific parts of application.
[6] Asynchronous module definition (AMD) [8] is completely modular architecture that allows reuse of 3rd-party components as well as reuse of all classes within application. All the classes are loosely-coupled and are not available through global identifier.
[7] Synchronizes model data with data displayed by the view. Also called two-way binding [37].
[8] Component controllers/view-models can only see and work with related data and components.
[9] HTML5 custom elements [12] allow creation of custom HTML elements and components without the need for additional JavaScript code or compilation to JavaScript.
[10] Internal component structure (incl. document object model — DOM) is not accessible from outside. See Shadow DOM [1].
[11] E.g. customizable finite-state machine within each component.

- Visual components are easily comparable between framework [3], many users select a framework based on the number or quality of visual components.
- Only experienced programmers think of client-side web applications as of desktop applications. Many programmers assume that architecture is not that important and that interactions can be "hardcoded", therefore choose simpler solutions — often only libraries like jQuery. The need a for good architecture raises over time with increased business logic complexity but the framework or the library cannot be easily changed at that point.
- Some frameworks were initally only a collection of visual components and evolved to a framework over time. Now, with a large user base, the original concepts could not be changed without losing the community.
- Visual components and visual appearance can be easily user for marketing, that is why commercial frameworks focus on them.

Some exceptions with interesting archutectural concepts exist and can be used as an inspiration:

- **AngularJS** which is partially service-oriented [34], services and components are *loosely-coupled* [39] and services are used to extend the component functionality. However, services cannot be used generally, they act more like framework modules.
- **Dojo** with aspect-oriented programming [26] features that can be used to extend functionality of any object, class or group of classes, e.g. to perform security checks on any object or to verify inputs and handle exceptions within any object method.
- **Meteor** which is purely event-driven [18] and events can be used e.g. for easier interaction testing without the simulation of DOM events (see the section 2.4.7). However, Meteor cannot be easily used with any back-end, it is not pure client-side framework.

2.1.2 Implications for the framework

The new framework is worth the creation, because only few frameworks have some abstract architecture and if so, it is one-way oriented. Additional, not yet covered business features from the Table 1.1 could be also included. Following observations were extracted from the framework analysis.

Focus on abstract architecture. The newly designed architecture could be compatible with service-oriented architecture or event-driven architecture and could include MV* support for components and possibly other software architecture patterns not yet present in the web frameworks.

Skip visual components. The choice of Dart with Polymer brings the background for creating visual components and the new framework does not have to deal with it. UI components can be therefore skipped — other frameworks have many of them, they could be reused with simple adaptors and Polymer can be used if user needs to create own ones.

Communication through libraries. Object-relational mapping, remote communication standards and formats can be also reused from existing libraries.

Skip language enhancements. Many frameworks try to solve class-related functionality instead of using specialized libraries or leaving the issues to a programming language that compiles to JavaScript, like Dart or CoffeeScript. The future version of ECMAScript, codenamed Harmony, will also contain class support [5]. Do not try to improve language imperfections even if some constructions might do the job better. The improvements may come over time by the Dart authors or there is a good reason why they were not included.

2.2 Architecture

The architecture will be designed by continuous improvements when dealing with goals and challenges from the chapter 1.

2.2.1 Architecture design

Mentioned already in the section 1.2, there is a need to anticipate code mistakes, especially those that are hard to detect and debug. In web application development (and especially in JavaScript) these include:

- Components can access any DOM element and can accidentaly change a DOM subtree that belongs to another component. This could be prevented if all the DOM manipulation was done through single point where component scope could be checked.
- Components can also access other application components and invoke ther internal methods or change their state from outside, because classes and instance objects are not protected or encapsulated in JavaScript. Components should have access only to related components [39] and references to them must be obtained from single, managed point to enforce this. Native encapsulation is solved in other languages (Dart, CoffeeScript) and in the future version of JavaScript [5].
- When working with components, programmer needs to deal with the component lifecycle — e.g. needs to check if it is fully functional or whether a visual component has its DOM rendered. When the component is destroyed, other components need to stop working with it. It is easy to forget verification of current component lifecycle phase and in combination with directly referenced components it could lead to code execution on a component with invalid or unexpected state. One solution is to work with components completely indirectly, without obtaining reference to them, e.g. using events.

The Figure 2.1 shows modular, AMD compliant architecture [8]. It includes single point for DOM access (part of the Framework Core). The references to other components (here called modules) are obtained indirectly from the module manager by their name; modules are not available in the global namespace [8]. However, once component reference is obtained, the modules communicate directly. This concept allows only static change of modules before an application is launched — once module is referenced from within other module it cannot be safely replaced.

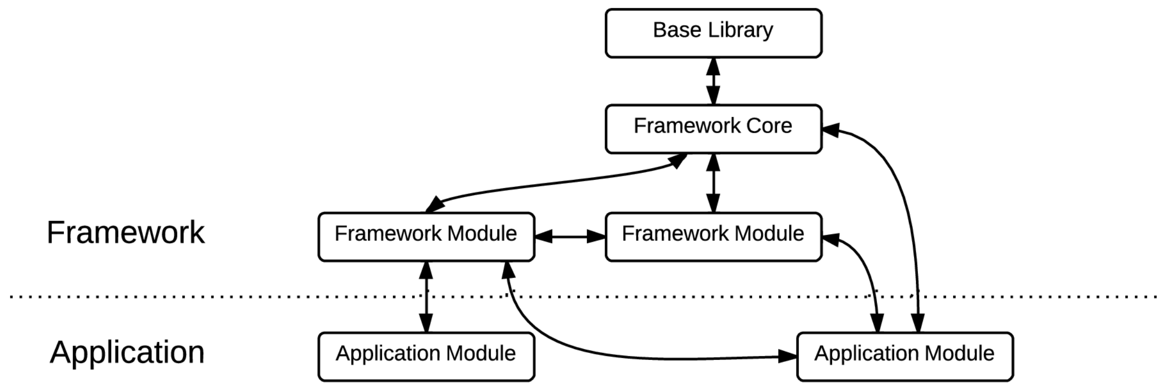


Figure 2.1: Modular architecture example.

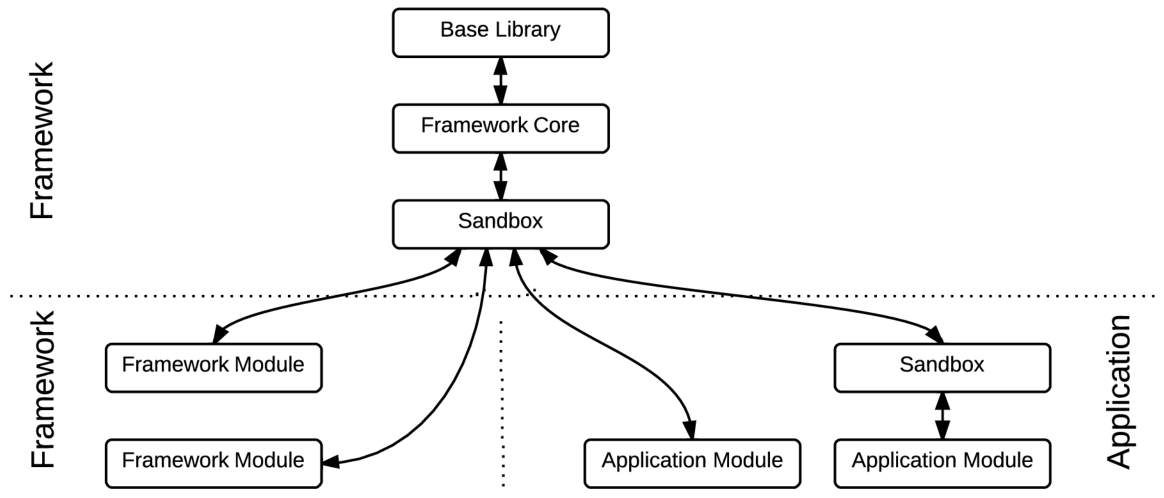


Figure 2.2: Scalable architecture example.

The extension of the modular architecture, an architecture that is fully scalable in terms that components never communicate directly and can be therefore dynamically replaced, added or removed, was suggested by Nicholas Zakas [38] and is visualized in the Figure 2.2. The modules do not know about any other application module and all the communication should flow through the sandbox. There are some drawbacks, though.

- The sandbox is user-defined and it should contain adaptor methods for all the features from base libraries [38]. If defined in wrong way it will contain hundreds of methods from libraries and many more for manipulation with application components. The proper way is to implement only publish-subscribe pattern [23] for communication and few methods for working with framework core. Even if implemented correctly, the sandbox API may not stay consistent with future application changes, because the proper implementation is not enforced in any way.

- While restricting component scope within specific application part, sandbox may be extended and duplicated for this purpose (as showed on the second Application Module). The functionality is then split into multiple objects and this inconsistency could lead to unintentional errors.

- Component permissions are not distributed uniformly. Framework core components have direct access to all the components while framework modules and application modules can communicate only through the sandbox [38]. The framework core is not supposed to reference components, but it is not enforced and it is possible source of errors.

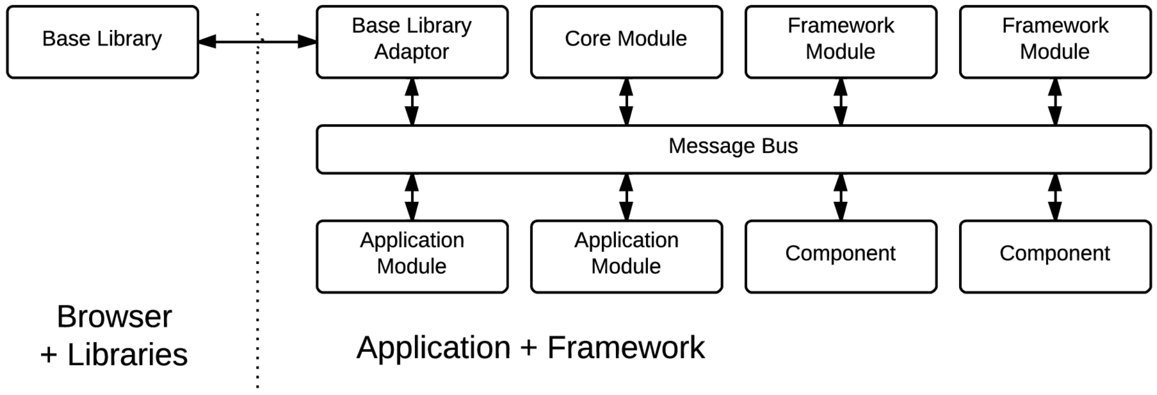


Figure 2.3: Message Bus architecture example.

The architecture that would solve previously analyzed problems is inspired by advanced software architectures, e.g. service-oriented architecture or Linux parts. These architectures have one element in common — a message bus for universal exchange of any piece of information called message. In Linux the bus is called D-Bus and is designed universally, allowing e.g. point-to-point communication, addressing, publish-subscribe communication, remote method invocations, . . . [2]. SOA uses enterprise service bus for indirect service invocation and gathering the result [34]. The Figure 2.3 shows that all the framework and application components are equal and that the framework and application implementations overlap. It is therefore easy to replace any part of the framework by an application-defined component. The architecture is flexible, modular and scalable, yet very simple. All the components are loosely-coupled and communicate indirectly.

Example of modularity and loose coupling advantages.

In an email client (Application 2) offline mode will be supported with all the basic functionality — writing emails and saving them to be sent once Internet connection is detected again. User will also have access to all previously opened emails and could perform search, delete or mark as spam operations. In other words the email behaves the same as in online mode, just works with lower number of records in the database.

The implementation is very simple within this architecture. Component providing access to the database at server side is replaced with component working with browser local storage when offline mode is detected. Both components have the same API (they provide the same services) and because they are not referenced directly anywhere in the application, the replacement is completely seamless [38]. No application component could even register the change, so everything works as usually and no special conditional statements for the offline mode are needed in the code.

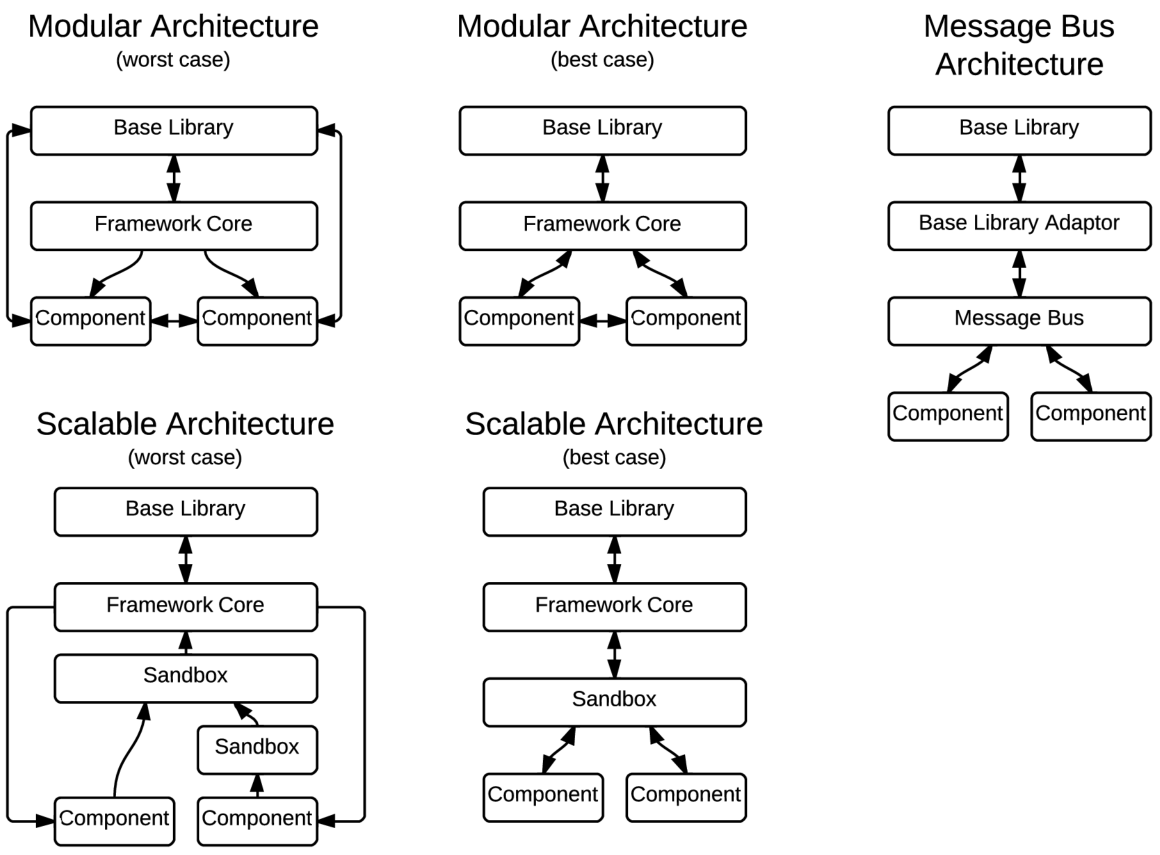


Figure 2.4: Comparison of communication flows within different framework architectures.

The Figure 2.4 demonstrates differences in the communication flows within these architectures. Modular and Scalable architectures do not have strict rules and user has more options how to implement data flows; shown are the best and the worst case scenarios. In the Modular architecture, the components have to communicate directly with each other even in the best case scenario. The scalable architecture has the same benefits as the Message Bus architecture if used properly. The Message Bus architecture has always only one possible way of implementing communication flows.

2.2.2 Message bus security

The security on the message bus was not addressed yet. Not all components should have the same application scope and be allowed to send or receive every message type (it is one of the framework business requirements mentioned in the Table 2.1). Three types of components have their behaviour embedded by default in the Message bus.

- **Managers** have access to all the messages on the bus.
- **Providers** can provide services available application-wide.
- **Normal components** can publish events but provision of services is forbidden. By default they have also access to all the messages on the bus but the access should be restricted by application-specific security rules. E.g. components could get access to all the messages of providers and managers, but not to the events of other components.

Custom security rules should not be implemented directly withing the bus, it should be barely extended application class to stay well-tested and optimized. Instead, managers that have access to all the messages on the bus should control the flow. Any "standard" message can be firstly encapsulated into a special security check message that only managers have access to. Every manager will then have the possibility to reject the standard message or modify its metadata before it enters the bus. However, managers do not have an option to allow the once-rejected messages re-enter the bus. Every message is allowed by default and managers could only restrict the rules.

Example of security rules logic.

In the project management system (Application 3) every component has different application scope. Every component can access specific range of domain objects and specific set of application services. The security rules are defined in a security manager that controls all the messages on the bus and by cooperation with other managers limits and filters access to the services and domain data.

The huge benefit is that components have no clue whether there are some security rules or not, the security manager is completely invisible to them. Security mechanisms are therefore not implemented inside components, which are designed more general in return. The shift of logic is nicely illustrated on a small example. Normally the component logic would be "Am I allowed to do this? If not, hide this element.". With the security rules implemented outside, the logic is shifted to more general level — "Is this service available in the application? If not, hide this element."

2.2.3 Internal communication

Support for specific communication patterns often used in web applications is implemented by default in the bus, because messages are too general concept. User has an option to implement other custom communication patterns by extending the bus class. The default are:

- **Events.** The global publish-subscribe pattern [23] for event notifications (either from DOM elements or from components generally).
- **Library functionality.** Second is the synchronous service request and response, used to get instant access to the functionality of base libraries. E.g. DOM query message with the result — DOM elements — instantly available to be used.
- **Services.** Last is universal asynchronous service request and response that could be used for external communication or for in-browser tasks that last longer, e.g. rendering of many DOM elements. Technically this could be achieved with synchronous behaviour where result promise [7] is returned instead of actual result and the promise is fulfilled later in the future.

The *global* publish-subscribe has one more useful feature — it prevents memory leaks. Global management has access to both publisher and subscriber and can automatically unregister event handlers when some of these components is destroyed. When publish-subscribe is implemented locally and event handlers are stored within publisher, all the event handlers need to be unsubscribed manually, because the publisher only knows the handler function but not its scope needed to unregister handler automatically.

If the subscriber is destroyed without unsubscription, the handler is still present. And because the handler still sees the original subscriber object in its scope, it prevents it from being destroyed by the garbage collector. The situation is even worse when publisher is application singleton that never gets destroyed. In this case millions of objects could be prevented from being freed, which could lead to very slow browser response or even crash caused by insufficient memory available. All is demonstrated in the Listing 2.1.

Listing 2.1: Memory-leak illustration within publish-subscribe pattern (pseudo code).

```
/// Subscriptions are stored within the publisher instance.
class Publisher {
  constructor: function() {
    this.subscriptions = [];
  },
  subscribe: function(event, handler) {
    return this.subscriptions.push([event, handler]);
  },
  unsubscribe: function(subscription) {
    this.subscriptions.remove(subscription);
  }
  // ... code for publish, destroy
}
var global = new Publisher(); // a global object

/// Class that subscribes to the global object event.
class Subscriber {
  constructor: function() {
    global.subscribe('event', this.handleEvent.bind(this));
  },
  handleEvent: function() { /* ... */ },
  destroy: function() { /* forgotten unsubscribe !!! */ }
}

/// Create and destroy many Subscriber instances.
for (var i=0; i<1000*1000; i++) {
  var subscriber = new Subscriber();
  subscriber.destroy(); // subscriber is never freed !!!
}
```

2.2.4 Architecture compatibility

Service-oriented architecture consists of independent applications that provide functionality (services) to other software parts. It was designed primarily for communication between applications from different vendors or using different technologies [34], but the concept can be easily applied to internal communication within single application. In the framework all the communication goes through the Message bus and messages are addressed indirectly. Therefore the components are independent on each other, can be replaced with custom or 3rd-party ones [38] (directly or with a simple adaptor). The components can be even written in a different language or technology, e.g. DOM/HTML events are easily converted into framework-specific messages. Framework architecture is therefore compatible with service-oriented architecture concept.

Event-driven architecture is an architecture where all the changes within component state are published through events [18]. Because of the framework architecture, the components cannot reference each other and to detect component changes programmer can either repeatedly test their state with special services or expose the changes through events. The latter one is preferred and if the programmer decides to expose all the component state changes as events, the application would be fully event-driven.

2.2.5 Architecture overview

By designing an architecture that would prevent errors, a scalable, modular and flexible architecture was created. User is able to replace any component from the framework core, can extend framework functionality by creating new globally available managers and can define own types of communication within the Message bus. Architecture is compatible with event-driven architecture and service-oriented architecture concepts. Created applications have lower probability of having serious DOM inconsistency errors or serious memory leaks.

2.3 Components

One of the ways to improve code clarity and to prevent mistakes, is to design a simple skeleton of every application component and include few rules that each component must follow. One of the problems in applications generally is that components can be very complex and therefore the class containing component logic and interactions implementation can be long and chaotic. That happens because the implementation is typically included within a single class. There are already some concepts that separate classes like MVC [36], but it is not sufficient. The main objective is to split the controller-like class into multiple classes that would contain the same-purpose functionality [39].

2.3.1 Configuration

Components, and especially visual components, could have hundreds of configuration properties[1] that blend with state properties, internal properties, methods etc. All the configuration properties should be moved to a separatate class [38].

In many frameworks there is unclear border between configuration properties and state properties or event subscriptions. Listing 2.2 shows the problem. The configuration is used to initialize a state variable, but uses the same name as the state variable which is confusing. Beause the configuration is applied directly to the instance object, it effectively replaces the state property value and therefore the original value is not available later when needed. Configuration also contains a method that influences component logic, but the behaviour should be implemented by extending the component class instead.

The configuration should contain only immutable values [21] and methods without component logic as shown in the Listing 2.3. To enforce immutability, framework locks for changes every configuration object when its component is initialized. As a side effect, **components with the same configuration will always behave the same** during their lifecycle, if placed into the same environment. This happens because the configuration is the only object that contains instance-specific values that can be influenced from outside[2].

[1]E.g. the basic `TextField` in ExtJS 4.2.2 has 27 own and 113 inherited configuration properties.

[2]Note that component behaviour cannot be changed by public instance properties, because programmer should not have access to these properties since the components are not referenced directly.

Listing 2.2: Misuse of configuration properties (pseudo code).

```
// Field constructor applies the configuration directly
// to the instance object.
var field = new TextField({
  // 'value' used as a configuration property.
  value: 'Initial value',
  // Event callback passed as a configuration property.
  onChange: function() {
    // 'value' used as a state property.
    if (this.value === 'Some value') {
      // Original value is not available.
      this.value = 'Initial value';
    }
  }
});
```

Listing 2.3: Correct usage of configuration properties (pseudo code).

```
// Field stores the configuration in a separate `config` object.
var field = new TextField({
  // Configuration property has different name.
  initialValue: 'Initial value',
  // This method has no influence on behaviour.
  formatValue: function(value) {
    return value + '...';
  }
});

// Event callback not passed as a configuration property.
field.on('change', function(field) {
  // State properties change does not influence configuration.
  if (field.value === 'Some value') {
    field.value = field.config.initialValue;
  }
}
```

Summarized, the locked configuration object is a barrier when programmer tries to create components with the traditional perception of components as complex and fully-featured systems. As a consequence it **forces programmer to split complex components into several fine-grained components**.

Example of components wrapping.

The Email client (Application 2) contains form for creating emails with input fields. All the fields should contain field labels that should change the displayed text when application language changes. Field label is not part of the text field state, so it is often moved to the configuration. But because configuration is locked, it cannot be dynamically changed. The solution is to split the component into multiple components that would wrap [23] each other — a field label and the actual text field in this example. Label component will be later replaced with a new instance without affecting the text field.

2.3.2 Component API

Components communicate heavily on the message bus and for an efficient distribution of messages, each component should expose its public API [34] that lets the message bus know in advance which services the component provides or which events publishes. Similarly to the configuration object, the API is locked after component creation and is therefore immutable during component lifecycle.

API is a separate object that removes communication-related definitions from the component class and improves clarity of the code. Because the API contains all the information about the way component communicates on the bus, programmer can find out the basic functionality and component behaviour only by examining its API, without the need to dig deeper into implementation details.

2.4 Framework extensions

The component structure described in the section 2.3 is fixed and contains only fundamental concepts. Other features should be implemented as component extensions, letting programmer choose the ones to use. This section describes interesting ideas that can be easily implemented thanks to the framework architecture and are useful for complex applications. The technical solution of component extensibility is described in the section 3.3.9.

2.4.1 Component state

Component state is a set of instance properties (including private ones) that are being changed during the component lifecycle. The state exactly describes difference in behaviour of newly created component and component with the same configuration in any phase of its lifecycle. If all the state properties were moved into a separate state object, it could be easily stored, recovered or replaced dynamically. Figure 2.5 shows the difference in component configuration object and state object roles.

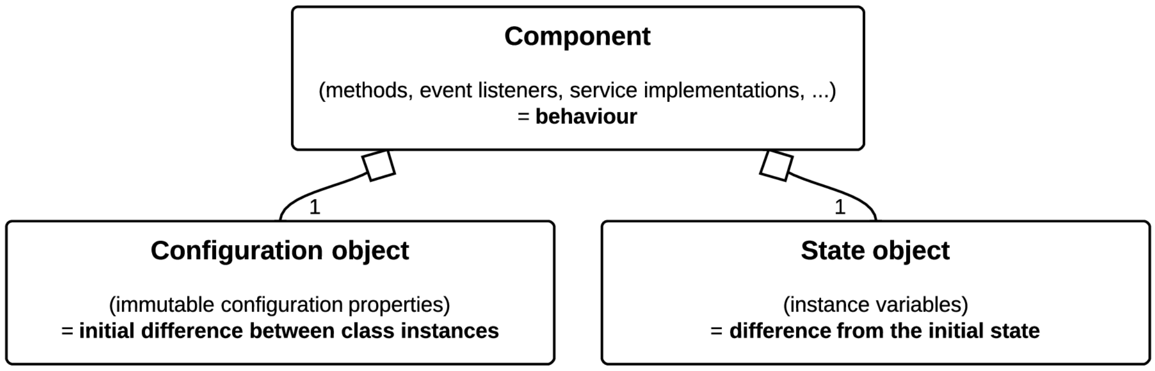

Component
(methods, event listeners, service implementations, ...)
= behaviour
1
1
Configuration object
(immutable configuration properties)
= initial difference between class instances
State object
(instance variables)
= difference from the initial state

Figure 2.5: Roles of component and its nested objects.

Separating state object brings many possibilities for working with components, e.g.:

- Push existing component into any lifecycle phase by replacing its state object.
- Continuously persist the config and state objects of all application components. When the application is reloaded, recreate the components with stored config objects and recover the application state and behaviour by replacing their state objects.

- Synchronize instances of the same application opened on more devices by synchronizing the state objects over HTTP. This could be useful e.g. for implementing applications supporting online teamwork or for live presentations where all participants have the application opened on own device, but it is controlled by the presenter.

Although this concept brings interesting opportunities, there are some drawbacks and this is the reason why it should be implemented as an optional component extension:

- It forces user to stop using instance variables within components. It is hard to keep this requirement, especially when a single instance variable present would violate the concept.
- Static variables are not included in the state object and are not stored anywhere. Programmers would have to stop using them within components. If they are used, components could have different behaviour even if recovered with the same configuration and state objects.
- The state object is valid only for a specific configuration object. If a state object is pushed to differently configured component, it might turn into invalid state.
- If components have application dependencies, framework would have to validate that they have mutually compatible states when state within some of these components is replaced or recovered. Ideally states of all dependent components should be recovered at the same time.

2.4.2 Data

One of the implications in the section 2.1.2 was to leave implementation of data objects and communication to preferred libraries of the programmer. Despite, the framework should establish a common way how to percieve the different data objects and where and how should they be used. The figure Figure 2.6 shows usage of the data objects in a sample application (even at the server-side). The three commonly used types of data objects are:

Domain objects (also called Business objects) model entities of the real world and typically map the database tables to classes. They are available and used by the whole application so they should contain meta-data, methods and transformations related to the application domain only [15].

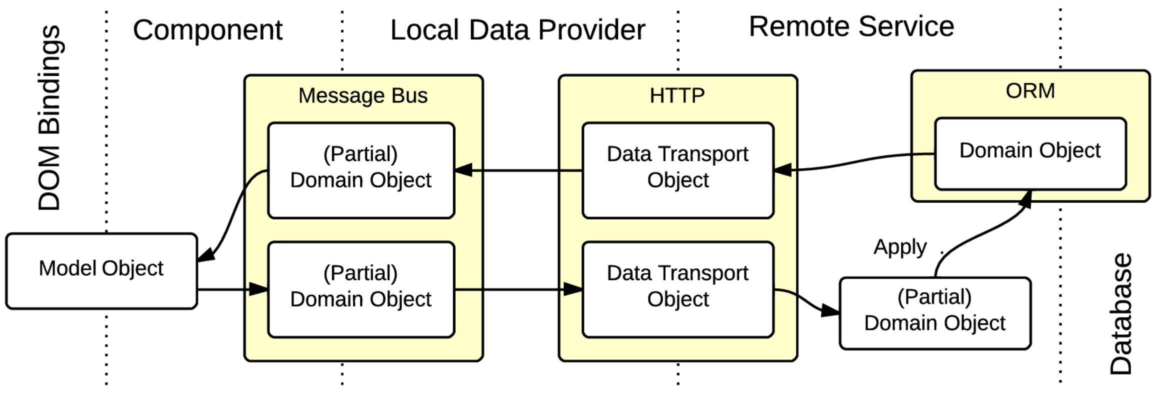


Figure 2.6: Example of data objects usage.

Data transport objects (DTO) are special objects used to transfer domain objects [15]. They can contain only partial information about domain object or contrarily can group multiple domain objects including the related ones into a single DTO [15].

Model objects are specialized views on domain objects. They are specific for each component and should simplify handling of domain objects enough to be easily used within the component. They can contain methods, computations and transformations related to the component. One model object can be created from multiple domain objects.

In applications created with the MV* concept, the domain objects are used directly in components [3]. This is all right if the views are simple enough to work with the domain objects directly. The problem is with the teminology because the term *Model object* is vague and the domain objects are often confusingly called *Models* [13]. The model object definition in this thesis is significantly different, as shown in the following example.

Example of data objects usage and transformations.

In the CMS system (Application 1) a grid with articles is displayed. The server sends a list of 20 displayed articles along with their authors and categories, all packed within one DTO. The `Domain Data Provider` parses this object back into multiple domain objects (articles, authors, categories). For usage within the grid, not all the information from domain objects are necessary, therefore a new, flat model object is created. It contains only few values matching the grid columns, which are already formatted and ready to be directly rendered. This model object is then bound [37] to the grid template.

2.4.3 Visual components

From the framework point of view, visual components (UI components, GUI components,graphical components) are exactly the same as any other component. They might only use services of DOM-related providers more often. The reason is that framework was designed with the idea that existing visual components from other frameworks or libraries can be reused (see the section 1.2).

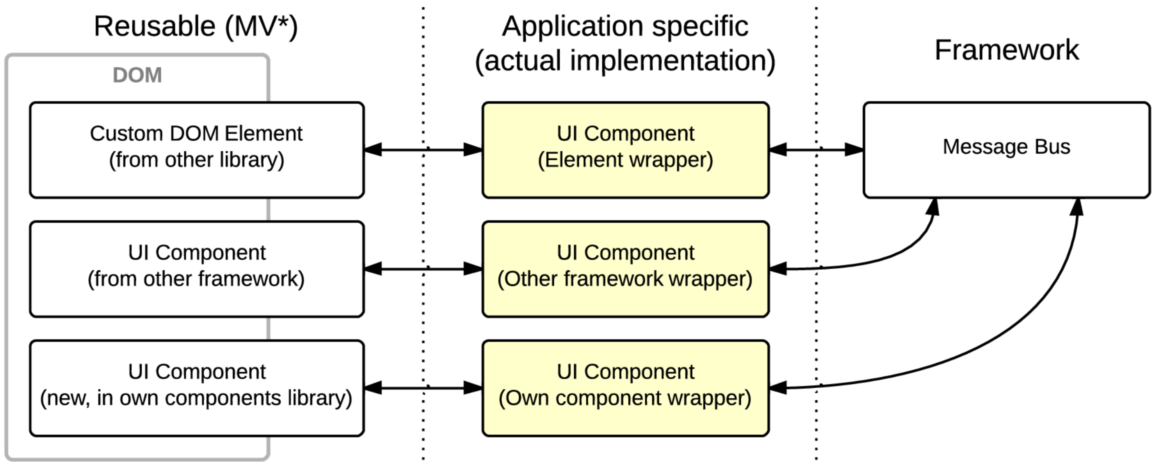

Reusable (MV*)
Application specific (actual implementation)
Framework
DOM
Custom DOM Element (from other library)
UI Component (Element wrapper)
Message Bus
UI Component (from other framework)
UI Component (Other framework wrapper)
UI Component (new, in own components library)
UI Component (Own component wrapper)

Figure 2.7: Visual components structure.

In practice, for every existing visual component, or custom DOM element, programmer creates a simple wrapper [23] that exposes component events and state to the Message bus as shown in the Figure 2.7. If needed, the component also includes application-specific behaviour. Note that in the Figure 2.7 the component wrappers do not access the DOM, not that it would be forbidden, but there is no reason why, because the DOM-related implementation is already done by the MV* part. Wrappers can then focus fully on application-specific behaviour. Completely new, custom visual components should be also created as reusable ones, in a separate library, together with separate application-specific wrapper classes.

2.4.4 Logical hierarchy

Visual components already partially belong to the DOM elements hierarchy. However, this hierarchy can be too complicated or can have different meaning than is the purpose and behaviour of a component. Additional logical hierarchy that is independent on the DOM should be present and the main benefits of its usage are:

- Also non-visual components can be included, compared to the DOM hierarchy.
- Child components can be automatically destroyed, which prevents memory leaks if programmer forgets to destroy them manually.
- Component can get equal access to logical child components that are placed in different DOM subtrees or containers. DOM hierarchy destroys only the DOM elements, not the component instances.
- Component might be allowed to control or subscribe to children messages on the Message bus, which is useful e.g. for exception handling [38].

Example of logical hierarchy usage.

In the Project Management system (Application 3) a detail view of a Task is displayed. It is a form with many fields split into multiple groups and panels. These containers prevent fields from being direct children of the detail view in the DOM. A logical hierarchy containing fields as direct children is created, completely ignoring containers which are not important for application logic. Code that is written with this hierarchy stays valid even if fields are reordered or put into different containers.

Additionally all the fields have labels that wrap the field (see Example of components wrapping). In DOM the labels are field parents but logically they should be field children. It makes even more sense when label is dynamically replaced with other language label — if label was a parent of the field, the field would be destroyed along with the label and would need to be recreated with the new one. This is not an issue when logical hierarchy is used.

2.4.5 DOM access

Reusable visual components should already have access only to the DOM elements they create. However, application components have generally unlimited access to the DOM. Ideally they should not use DOM at all, but when they need to for some reason, security rules should enforce that components do not accidentally change a DOM subtree that is not logically connected to the component. Therefore a DOM manager should be present. It will:

- Serve as a hub and the only point in the application for accessing and manipulating DOM elements [38].
- Restrict access to the DOM based on application-specific security rules.

2.4.6 State machine

It would be nice to describe component lifecycle by a finite-state machine, state transitions could be then published as events and other components could react to these state changes. Transitions could be also done automatically by reacting to the messages on the bus. The problem is that the component lifecycle can be very complicated and cannot be described by a single flat finite-state machine without excessive redundancy. To fully cover all the possible states without any redundancy, these structures are required:

Hierarchical, concurrent finite-state machine where states can contain substates within multiple, independent branches [19]. This state machine therefore contains multiple active states, but all of them are covered by a single state from the top level state machine.

Asynchronous dynamical state chains that consist of a sequence of predefined states that can be included multiple times. These dynamical states are independent on the main state machine.

The possibilities are described on the example in the Figure 2.8. The component contains the top-level state machine (`Initializing`, `Running`, `Destroyed` states) and the Running state is split into multiple state machines defined by classes and mix-ins in the class hierarchy, together forming one hierarchical state machine [19]. During the `Running` phase two

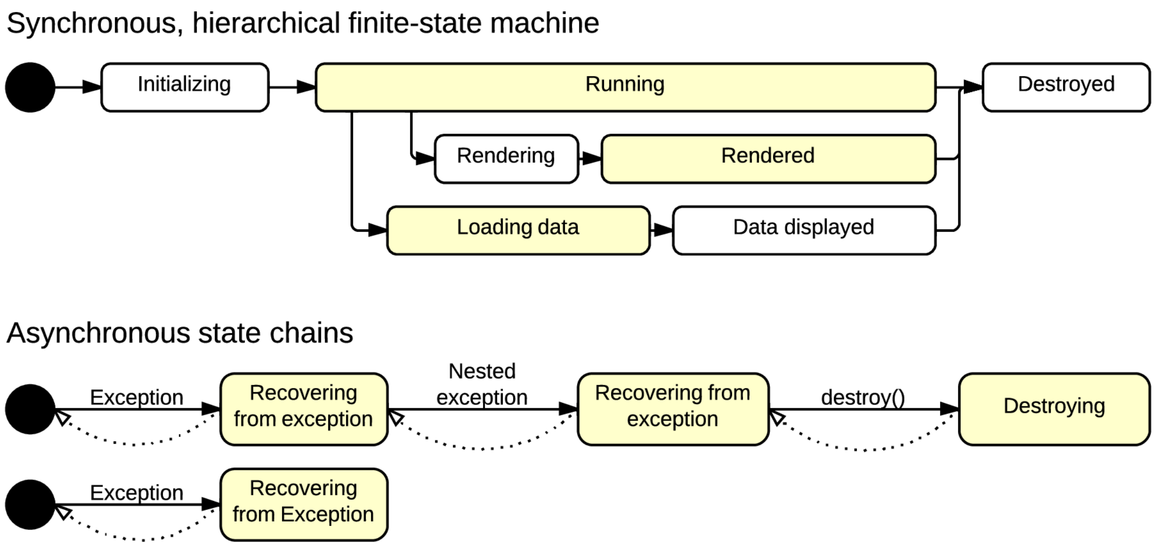

Synchronous, hierarchical finite-state machine
Initializing
Running
Destroyed
Rendering
Rendered
Loading data
Data displayed
Asynchronous state chains
Exception
Recovering from exception
Nested exception
Recovering from exception
destroy()
Destroying
Exception
Recovering from Exception

Figure 2.8: Example of simultaneous component states.

independent exceptions occured, each creating new dynamical state chain. These chains are independent on the synchronous state machine, because they could be reversed. E.g. the `Destroying` state can be reverted if a superior component prohibits the component from being destroyed. The `Recovering from Exception` state can be reversed if the component or superior component has ways how to cope with the exception. Meanwhile recovering or destroying the component should logically stay in the original states from the main state machine. In the example the component is in 7 states at the same time, 3 of which have the same name and meaning.

2.4.7 Testing

The fully event-driven architecture (see section 2.2.4) opens many extension options. It can radically change the way web applications are tested today. There are many libraries for unit testing [39] that is a great way to test classes, but it cannot easily capture user interactions [21][30]. For these purposes additional browser extensions or tools to simulate interactions were built[3]. They work by triggering fake DOM events and then verify the new DOM structure [29]. Still the programmer needs to install additional software or plugin and if the DOM structure or names are changed, tests need to be rewritten.

Framework architecture enables interaction testing indepentendly on the DOM structure, without any additional tool. The prerequisity is that all the components are fully event-driven — to repeat what was stated in the section 2.2.4, it means that every state change (even internal) is published on the message bus. Three possible ways of interaction testing shown in the Figure 2.9 are inspired by testing patterns within service-oriented architectures [17]. They complement each other — ideally all the test types should be created for every application module, along with unit tests.

Isolation tests. Only the tested component and the test case component are connected to the bus. Often the tested component cannot be fully isolated and needs to communicate with other components or service providers. Functionality of these dependencies can be then made available e.g. using mock objects that contain only partial or minimal implementation of the needed functionality [21][29]. Components can be tested in two ways:

- By simulating DOM events directly. The component should respond by publishing application events that correspond to the DOM events. This test verifies component reactions to the DOM event, but does not test the functionality or correct behaviour itself. These tests are dependent on the DOM structure.

- By simulating messages the component listens to. Component should respond by state change events and other application messages. This tests that component reacts correctly to changes in the environment — meaning that it changes its state in corresponding way, requests correct services and reacts with correct events.

Integration tests test multiple related components at the same time for their correct cooperation [17]. It is similar to the isolation tests, just works with multiple components. The test case should work with application events only as the reaction to DOM events is already verified by isolation tests.

[3]This type of testing is sometimes called End-to-end testing [29] and tools include Selenium [39] or AngularJS Protractor [29]

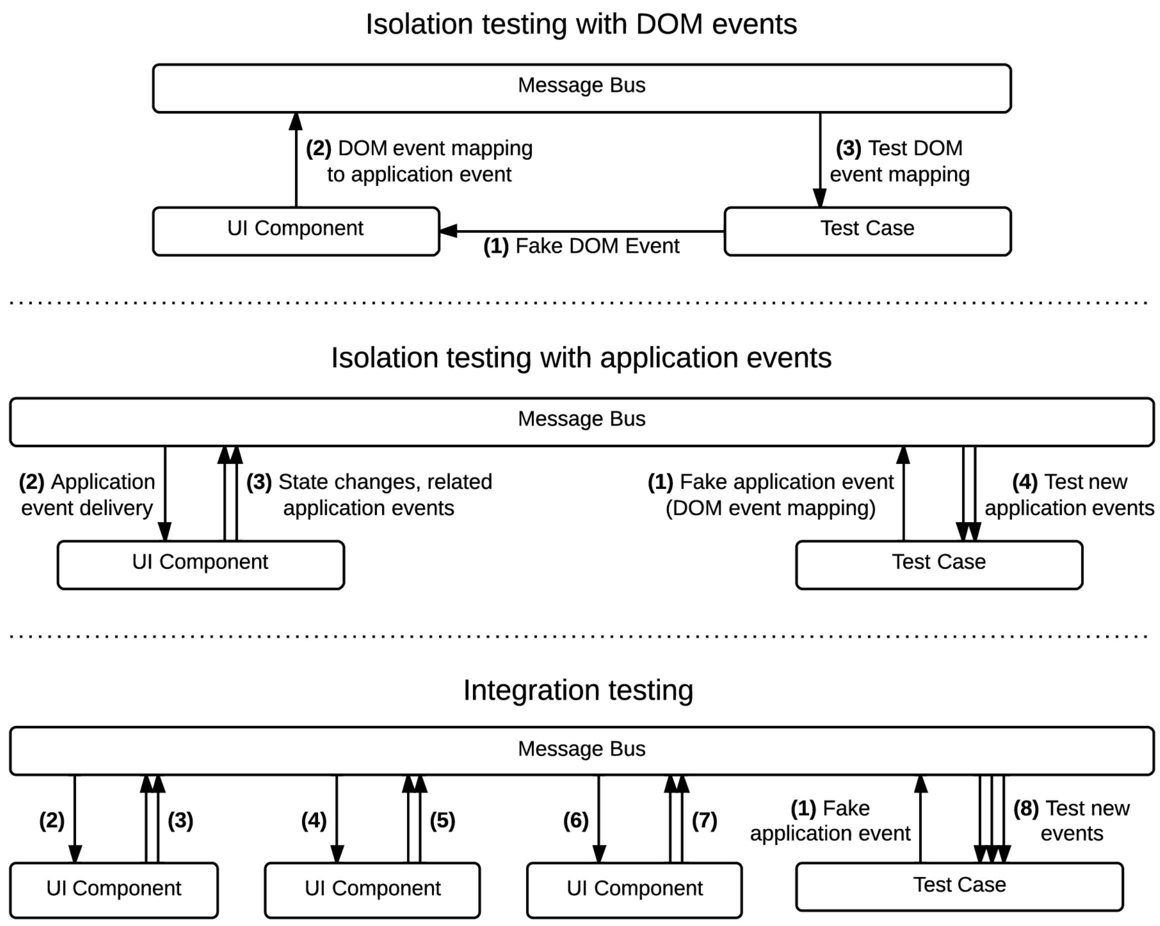

Isolation testing with DOM events
Message Bus
(2) DOM event mapping to application event
(3) Test DOM event mapping
UI Component
Test Case
(1) Fake DOM Event
Isolation testing with application events
Message Bus
(2) Application event delivery
(3) State changes, related application events
(1) Fake application event (DOM event mapping)
(4) Test new application events
UI Component
Test Case
Integration testing
Message Bus
(2)
(3)
(4)
(5)
(6)
(7)
(1) Fake application event
(8) Test new events
UI Component
UI Component
UI Component
Test Case

Figure 2.9: Three ways of testing component interactions.

2.4.8 Exceptions handling

For more precise handling of exceptions, they can be wrapped into special message type and processed at multiple levels by components listening on the bus. Exceptions can be for example passed up in the logical hierarchy tree unless processed, then passed to local or global exception handlers (special manager components) for logging.

Chapter 3

Technical solution

This section covers technical and implementation details of the framework architecture, selected extensions and technologies. The base technologies used have some limitations or preferred ways of solving problems and may not allow implementation of the framework in the full extent, as it was analysed in the chapter 2.

3.1 Base technologies

The **Dart language** was chosen as the base implementation language. It is new, modern language that has a goal of being easy to undestrand by average programmer while adding numerous benefits over currently used languages. The ones with huge impact on this framework and applications development generally are:

Native class system. Many frameworks focus on creating own class system that is not present in JavaScript. With Dart all the class-related language features like classes, mix-ins, interfaces, static fields, annotations, generics, . . . are present. The class system also ensures that instance variables are not shared accross class instances, which is possible in JavaScript-based class systems and which causes hard-to-find errors as shown in the Listing 3.1:

Listing 3.1: Example of unexpected behaviour caused by a shared instance variable.

```
Class.create('Container', {
  // The programmer has forgotten (or does not know) that
  // Arrays and Maps defined in the prototype are shared.
  children: [],
  title: ''
});

var first = new Container();
first.title = 'First';
first.children.push('Child 1');

var second = new Container();
second.title = 'Second';
second.children.push('Child 2');

// Unexpected content of the `children` instance variable !!!
// first  => {title: 'First',  children: ['Child 1', 'Child 2']}
// second => {title: 'Second', children: ['Child 1', 'Child 2']}
```

Optionally typed variables. It is possible to quickly develop application prototypes without using type annotations and add them later when prototype is selected for production development and static analysis is needed. JavaScript does not have type annotations, so static analysis is extremely difficult, is adjusted separately for each framework and fails if application breaks framework rules.

Packages and libraries support is included in the SDK and package creation is recommended everywhere. This encourages programmers to structure applications in a better way and to create public or internal reusable libraries.

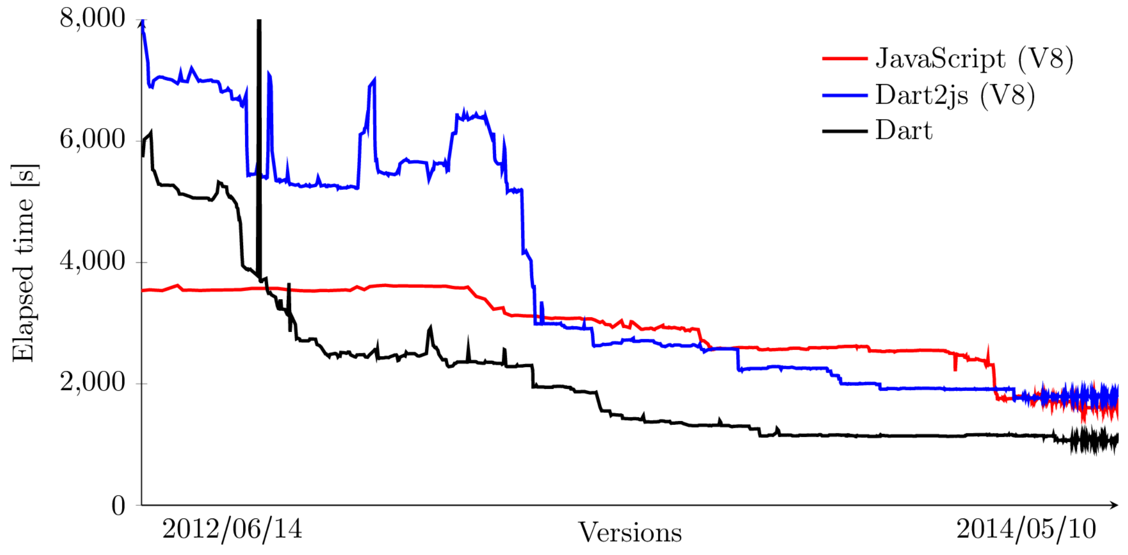
Two levels of objects visibility — library private and public. In other words every object within specific library has access to all the classes, objects and their fields, even private ones. When library is imported, only public objects are visible. This lack of truly private or protected objects forces programmer to encapsulate logically related classes together and create many small libraries in order to keep the security at a sustainable level. Therefore developers have to analyze applications in more details before programming.

Browsers and performance. Dart is a language created by Google, so it should support the same browsers that are supported by Google services. The promise is to include the latest versions of every major browsers and two latest versions of Internet Explorer [4].

Dart can be compiled to JavaScript with `Dart2js` tool and a special Chromium browser build, called Dartium, already includes native Dart Virtual Machine (Dart VM). The comparison of performance is visualized in the Figure 3.1, which shows one of the benchmarks used by the Dart team [14]. Dart language running in the native environment already outperforms JavaScript and more optimizations should come with new Dart SDK releases.

To involve other vendors to include Dart VM into their browsers, an ECMA technical commitee was established and is working on the Dart language standard. ECMA is the organization that standardizes JavaScript so it is anticipated that the vendors will join these efforts [27].

Figure 3.1: Dart performance comparison with the DeltaBlue benchmark [14].



Polymer.dart adds support for not yet implemented HTML5 features to all major browsers. Visual components can be then easily created with HTML5 Custom Elements API. See Table 2.1 for comparison of the features that Polymer brings.

3.2 Libraries structure

3.2.1 Framework

To make framework fully extensible, a small core library called `framework.core` was created. It contains implementation of the message bus architecture (section 2.2) and skeleton of a base application component (section 2.3).

Every extension to the core (see section 2.4) is also packed as a separate library. All the extensions are then imported and applied to the component in the `framework` package, but programmer always has a choice to import `framework.core` library instead and apply only the needed extensions, or add custom ones. Framework is therefore extremely modular and extensible, with the `framework.core` being the only fixed part.

3.2.2 Applications

Additional packages were created for easier implementation of web applications. The package called `framework_polymer` adds Polymer.dart support including its correct initialization. The `polymer_bootstrap` includes Polymer-compatible version of Twitter Bootstrap visual style and typography library. `Polymer_elements` contains set of reusable visual components used e.g. in the sample applications.

The implemented application can combine these packages as needed. The only requirement for each application is to create a custom class that extends `Application` core class. Its instance should be then manually launched. The implementation must also initialize the message bus and connect initial components. Some extensions also require initialization of related providers and managers.

3.3 Implementation challenges

3.3.1 Programming conventions in the framework core

Library private properties and classes are marked with `_$` prefix and class private properties with the standard `_` prefix. While analyzing class source code it helps to easily identify properties that are used elsewhere in the library and where more attention must be kept during refactoring.

Empty values are always passed as `null`. Empty strings, arrays or maps should be used only in special situations. This unites the arguments checks and assertions, because the values can be tested to null independently on the actual object type. It is also useful for data types that do not have empty value, like numbers. In JavaScript the `null` comparison is not recommended [39] because it does not verify object type. In Dart the situation is opposite, because the variables have type annotations.

Documentation in the Markdown format that is already supported by Dart SDK in a slightly modified form.

3.3.2 Messages

Messages that enter the bus should be deep immutable, only the metadata should be modifiable. However, Dart libraries contain shallow immutable collections only. The shallow immutable objects can be also achieved by `const` constructors, but then different instances created with the same data will be equal, which is wrong behaviour. Therefore the framework as of the current version does not support immutable messages and is up to programmer not to modify them.

The main challenge was to suggest a method to distinguish different message types. The problem can be demonstrated on events. DOM events contain `String name` property for this purpose. When a specific data have to be passed a subclass for the specific event type is created, but the `name` still has to be passed to the superclass which leads to unnecessary code duplication. Additionally it is not a good practice to write `String` values directly, so these should be stored as static class constants, which again increases code duplication and decentralizes event declarations as illustrated in the Listing 3.2. Code is then prone to copy-and-paste errors because programmer might forget to change some names.

Listing 3.2: Problems with events referenced by the `name` property.

```
class Event() {
  static const String CLICK = 'click';
  static const String MOUSE_DOWN = 'mousedown';

  final String name;
  Event(this.name);
}

class KeyPressEvent() {
  // Duplication of code -- `KeyPress` is already
  // present in the class name.
  static const String KEY_PRESS = 'keypress';

  final int keyCode;
  KeyPressEvent(this.keyCode, String name) : super(name);
}

// Decentralized event constants -- some stored in the [Event]
// class, some in the [KeyPressEvent] class.
Event event = new Event(Event.CLICK);
Event event = new KeyPressEvent(Key.ENTER, KeyPressEvent.
    KEY_PRESS);

if (event.name == KeyPressEvent.KEY_PRESS) /* ... */
```

For more efficient way the `String` type can be replaced with the `Symbol` instances that are faster, but it will not solve any of the previously mentioned issues. The DOM approach that is used in many frameworks and libraries is therefore not a good way to go. The solution is to reference messages directly by their class name. For every event type a new class must be present as shown in the Listing 3.3. This approach requires programmer to specify the message name only once and was applied to all the messages on the message bus.

Listing 3.3: Events referenced directly by their class type.

```
// Event class cannot be instantialized directly.
abstract class Event() {}

// Each event type has its own class.
class ClickEvent extends Event {}
class MouseDownEvent extends Event {}
class KeyPressEvent extends Event {
  final int keyCode;
  KeyPressEvent(this.keyCode);
}

// No constants needed, simply create the event.
Event event = new ClickEvent();
Event event = new KeyPressEvent(Key.ENTER);

// Events are distinguished by their class.
if (event is KeyPressEvent) /* ... */
```

3.3.3 Messages metadata

Application managers and providers need to know additional information about the messages, like who originated the message and when, and need to mark specific flags for other managers, etc. If these metadata were placed within the message, they would need to be initialized in the message constructor, because all the messages are immutable once created. Therefore directly the component that creates the message would need to deal with them, which would add additional burden on the programmer, plus the metadata could be easily faked. Ideally the message bus should take care of this, but when the message enters the bus, it is too late to change its properties.

Therefore a separate `MessageMetadata` object is created for each message. The benefits are that metadata cannot be faked, because they could be created only by the message bus, and that the actual messages contain only relevant data. The metadata are then linked to the messages with `Expando` class that creates loose association between objects and does not prevent garbage collector from freeing the contained objects. Message metadata are then returned by the message bus on-demand, they are not passed along with each message, as shown in the Listing 3.4.

Listing 3.4: Example of message metadata usage.

```
class SecurityManager extends Manager {
  // ... initialization etc.
  // Allows only messages from [Provider]s and [Manager]s
  // to enter the bus.
  SecurityCheckResponse onMessage(SecurityCheckRequest request) {
    Message message = request.message;
    MessageMetadata meta = bus.getMetadata(message);

    if (meta.source is Manager || meta.source is Provider) {
      return new SecurityCheckResponse(allowed: true);
    }
    return new SecurityCheckResponse(allowed: false);
  }
}
```

3.3.4 Components

The `Component` class is the base abstract class from which all components connected to the bus must inherit. The `Manager`, `Provider` classes and all the class extensions could be applied to every component as mix-ins. There is no default component class hierarchy — all the functionality is implemented as mix-ins. This approach allows programmer to create own base component class, specific for the developed application, by applying mix-ins at a single place. It leads to more extensible framework, because if there was a class hierarchy, mix-ins would have to be applied separately to each class. It is demonstrated in listings 3.5 and 3.6.

Listing 3.5: Framework with internal component class hierarchy.

```
// `framework` package
abstract class Component { /* ... */ }
abstract class Provider extends Component { /* ... */ }
abstract class Manager extends Component { /* ... */ }
abstract class UIComponent extends Component { /* ... */ }

// `application` - apply [Mixin1] and [Mixin2] to all components
abstract class BaseComponent extends Component
  with Mixin1, Mixin2 {}
abstract class BaseProvider extends Provider
  with Mixin1, Mixin2 {}
abstract class BaseManager extends Manager
  with Mixin1, Mixin2 {}
abstract class BaseUIComponent extends UIComponent
  with Mixin1, Mixin2 {}

// Create custom [Manager], not possible to have [Manager]
// and [Provider] at the same time, so 2 classes are created.
class SecurityManager extends BaseManager { /* ... */ }
class SecurityProvider extends BaseProvider { /* ... */ }

// Create custom UI component.
class TextField extends BaseUIComponent { /* ... */ }
```

Listing 3.6: Framework without component class hierarchy, functionality as mix-ins.

```
// `framework` package
abstract class Component { /* ... */ }
abstract class ProviderMixin { /* ... */ }
abstract class ManagerMixin { /* ... */ }
abstract class UIComponentMixin { /* ... */ }

// `application` - apply [Mixin1] and [Mixin2] to all components.
abstract class BaseComponent extends Component
  with Mixin1, Mixin2 {}

// Create custom [Manager] and [Provider] at the same time.
class SecurityManager extends BaseComponent
  with ManagerMixin, ProviderMixin {}

// Create custom UI component
class TextField extends BaseComponent with UIComponentMixin {}
```

3.3.5 Base component class

The listings 3.5 and 3.6 show custom base of every application component, called `BaseComponent`. It would be nice if programmer could use the name `Component` instead, meaning that the custom component implementation would hide the class from the framework core. It could be achieved by a specific `import` combination as shown in the Listing 3.7.

Listing 3.7: Extension of core `Component` class without the name change.

```
import 'package:framework/framework.dart' hide Component;
import 'package:framework/framework.dart' as core show Component;

// [Component] class hides the `Component` from framework core.
class Component extends core.Component with Mixin1, Mixin2 {}

// Create custom class based on the extended class
class MyComponent extends Component {}
```

This technique has one drawback demonstrated in the Listing 3.8. If a provider defined outside the application returns a component, it is of the `core.Component` type which may be accidentally assigned to the extended `Component` variable. This assignment causes exception because more general class is assigned to its subclass. Generally the service providers should not return components, because all the communication between components should be indirect. Still there might be good reasons to return components directly and programmers need to know about this issue.

Listing 3.8: Demonstration of the `Component` class name mismatch.

```
class Component extends core.Component with ... {}

// Service request that returns [core.Component] instance,
// but the programmer uses the [Component] type.
var request = new HirarchyParentRequest(this);
HierarchyParentResponse response = bus.send(request);

// Throws assignment exception.
Component parent = response.parent;

// This works.
core.Component parent = response.parent;
```

3.3.6 Component constructor

The constructor needs to be in a special form, so that static analysis can work correctly with the `config` instance variable. The Listing 3.9 illustrates the problem.

Listing 3.9: Correct and incorrect forms of Component constructor.

```
class MyComponentConfig extends ComponentConfig { /* ... */ }

/// Correctly typed config
class MyComponent extends Component {
  final MyComponentConfig config;
  MyComponent(MyComponentConfig config) : this.config = config,
      super(config);
}
```

```
/// Incorrectly typed config
class MyComponent extends Component {
  MyComponent(MyComponentConfig config) : super(config);
  // Static analysis considers this.config of [ComponentConfig]
  // type because the field is inherited from the superclass.
}
```

3.3.7 Component-related classes

The component implementation is split into multiple classes (see section 2.3) and framework contains a way to specify the class relations. These connections are important for static and runtime analysis that can detect copy-and-paste errors or wrong object usage more easily. For this purpose a `@PartOf` annotation is present in the framework. Static analysis benefits are shown in the Listing 3.10.

Listing 3.10: Static analysis errors detections with the `@PartOf` annotation.

```
// Correct definition.
@PartOf(MyComponent)
class MyComponentConfig extends ComponentConfig {}
class MyComponent extends Component {}

// Copy & paste error -- detected because [OtherComponentConfig]
// has @PartOf(OtherComponent) annotation instead of expected
// @PartOf(Component).
@PartOf(MyComponent)
class MyComponentConfig extends OtherComponentConfig {}
class MyComponent extends Component {}

// Copy & parse error -- detected because [MyComponentConfig]
// type is expected for the [config] field.
@PartOf(MyComponent)
class MyComponentConfig extends ComponentConfig {}
class MyComponent extends Component {
  OtherComponentConfig config;
  // ... constructor
}

// Even copy & paste error within the annotation is detected,
// because [MyComponent] has no related [Config] class.
@PartOf(OtherComponent)
class MyComponentConfig extends ComponentConfig {}
class MyComponent extends Component {}
```

The `@PartOf` annotation is used in the same way for the messages on the message bus. The `@Follows` annotation is also used for semantic analysis of message classes (and can be possibly used for any logically subsequent classes). The meaning is illustrated in the Listing 3.11.

Generally the services need to be manually registered to the component API (as explained in the section 2.3.2). If the `@PartOf` annotation is used, the related services can be registered automatically with framework transformers (Dart code pre-compilers), shown in the Listing 3.12.

Listing 3.11: Usage of the `@PartOf` annotation for message classes.

```
// Runtime checks verify that only [MyProvider] publishes
// [MyEvent] events.
@PartOf(MyProvider)
class MyEvent extends Event {}

// Runtime checks verify that only [MyProvider] responds to
// [MyServiceRequest] service requests.
@PartOf(MyProvider)
class MyServiceRequest extends ServiceRequest { /* ... */ }

// Runtime checks verify that provider always responds with
// [MyServiceResponse] to [MyServiceRequest] service requests.
@Follows(MyServiceRequest)
class MyServiceResponse extends ServiceResponse { /* ... */ }
```

Listing 3.12: Automatic API registration with `@PartOf` annotation.

```
// Provider class without the @PartOf annotations usage.
class MyProvider extends Component with Provider {
  // Register the service into [ComponentApi].
  MyProvider(...) : ..., super(...) {
    api.add(new ServiceApiElement(MyServiceRequest));
  }
}

// Provider class with services registered automatically.
class MyProvider extends Component with Provider {
  MyProvider(...) : ..., super(...);
}
```

3.3.8 Message subscriptions

To simplify subscriptions and reaction to messages on the bus, the `@On` annotation is present. It is applicable to any private instance method with specified format within component class. The method declaration format is different for every message type, e.g. `void _method(Event event)` is used for events and services handler method is the `ServiceResponse _method(ServiceRequest request)` format. The Listing 3.13 shows the usage.

Listing 3.13: Usage of the `@On` annotation to simplify message subscriptions.

```
// Manual subscription
class MyProvider extends Component with Provider {
  MyProvider(...) : ..., super(...) {
    // Register service callback.
    api.add(new ServiceApiElement(MyServiceRequest, _onService));

    // Event subscription.
    var request = new SubscribeRequest(OtherComponentEvent);
    SubscribeResponse response = bus.send(request);
    request.stream.listen(_onEvent);
  }

  void _onEvent(OtherComponentEvent event) { /* ... */ }

  MyServiceResponse _onService(MyServiceRequest request) {
    return new MyServiceResponse();
  }
}

// Automatic subscription
class MyProvider extends Component with Provider {
  MyProvider(...) : ..., super(...);

  @On(OtherComponentEvent)
  void _onEvent(OtherComponentEvent event) { /* ... */ }

  @On(MyServiceRequest)
  MyServiceResponse _onService(MyServiceRequest request) {
    return new MyServiceResponse();
  }
}
```

With the `@PartOf` and `@On` annotations it is **extremely easy to develop fully event-driven applications**. Component implementation will consist mostly of the message subscriptions. No public methods should be needed, because components are not referenced directly. The component implementation is therefore nicely encapsulated. See also Appendix C for a step-by-step example of application development where these annotations are used extensively.

3.3.9 Internal component messages

Different component states were already examined in the section 2.4.1, but the constructor phase should be analyzed in more details, because all the classes in the class hierarchy, including mix-ins, are initializing the component at that time. The framework-specific way of initialization is shown in the Figure 3.2. In the first `Configuring` phase, the component configuration is modified and locked for changes at the end. During the `Initializing` phase the component state, API and instance variables are initialized. At the beginning of the `Running` phase the component is fully functional, publishes its API publicly and starts communicating on the message bus.

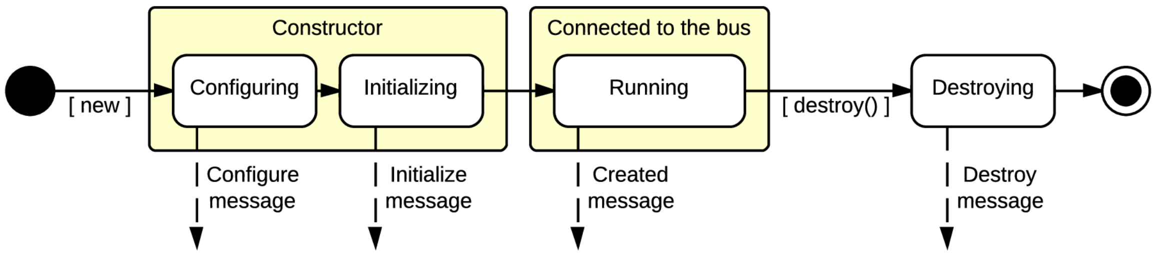


Figure 3.2: Visualization of private messages during component lifecycle.

The challenge is to allow every subclass and mix-in to run their own code during each phase. If only subclasses were present, it could be easily done by specially named methods (often called hooks). But the mix-ins in Dart cannot contain the same-name fields as classes they extend, so another solution must be chosen. Each state/phase could start with an internal component event, to which only subclasses and mix-ins could subscribe, as shown in the Figure 3.2. For subscription to events on the message bus the components already use the `@On` annotation, so if the internal events were tunelled through the message bus, it could be reused for this purpose. These internal events inherit from the `PrivateEvent` class and the message bus makes sure that they are not available publicly. Usage example is shown in the Listing 3.14.

Listing 3.14: Usage of the `@On` annotation for internal events subscription.

```
// Internal component events definition.
class ComponentConfigure extends PrivateEvent {}
class ComponentInitialize extends PrivateEvent {}
class ComponentCreated extends PrivateEvent {}
class ComponentDestroy extends PrivateEvent {}

// Component constructor in the framework core publishes events.
class Component extends Object with ... {
  Component(...) : ... {
    // Plug myself to the bus -- allow reaction to private events
    bus.plug(this);

    // Configure phase.
    sendMessage(new ComponentConfigure());
    config.lock();

    // Initialize phase
    sendMessage(new ComponentInitialize());
    api.lock();

    // Running phase, executed asynchronously.
    scheduleMicrotask(() {
      bus.publishApi(this);
      sendMessage(new ComponentCreated());
    });
  }
}

// Automatically registers components into a logical hierarchy.
class HierarchyMixin {
  @On(ComponentCreated)
  void _onCreated() {
    Component parent = config.parent;
    bus.send(new HierarchyRegisterRequest(parent));
  }

  @On(ComponentDestroy)
  void _onDestroy() {
    bus.send(new HierarchyUnregisterRequest());
  }
}
```

The difference to application events is that internal event subscriptions are not executed in the registration order but in the order from superclass to subclass. The destroy event is distributed in reversed order starting with subclasses. That way the class implementation can rely on correctly initialized functionality by superclasses.

3.3.10 Internal framework security

The framework should also contain internal security mechanisms that verify e.g. that components cannot fake messages, change message source, metadata etc. Some of these will be only available with the deep object immutability support, which is not present in the Dart language yet. The validity of messages can be easily verified only if all the messages are passed through single message bus access point. For these purposes the methods `sendMessage` and `getMetadata` are available in every component for working with the bus. The `bus.send()` and other bus methods referenced in previous code snippets are used only internally by the framework and are not publicly available.

3.4 Issues to be solved

3.4.1 Annotations and performance

The functionality of the `@On`, `@PartOf` and `@Follows` annotations could be implemented in two ways.

Using reflection. The component includes special mix-in that makes annotations work by checking annotations on objects dynamically during the application lifetime. It uses reflection, which has severe impact on performance. Measured experimentally by the throughput of the message bus in messages per second, the results in an application that used reflection were worse in magniture of orders compared to the application without reflection. Additionally the reflection is not currently supported in the `dart2js` compiler.

The method is, however, resistant to the changes in Dart language specification and to framework core changes, which is benefitial in the early framework development stage, where changes happen rapidly. That is why the annotations are implemented this way in the current framework version.

Using Dart transformers. Transormers modify the Dart source code before it is compiled or passed to the browser. Annotations are found by static analysis of the source code. The functionality is then added through generated code snippets put directly into the class implementation. Annotations functionality generated this way have no impact on the performance, because the reflection is not used.

The drawback is that user does not have control over the generated source code, so the transformers have to be error-free and well tested. Also, they have to be tested with every new Dart language version to verify that the generated code complies with the new specification. The reflection method should be therefore replaced with Dart transformers only when framework core and Dart language specification are stable.

3.4.2 Annotations and polymorphism

The methods annotated with the `@On` annotation should be private, therefore not even visible to own subclasses and should not be overriden. The problem occurs when two private methods have the same name, because the language polymorphism returns only one, the most specific, method implementation to be called. This is demonstrated in the Listing 3.15.

Listing 3.15: Usage of the `@On` annotation for internal events subscription.

```
// subclasses with the same name methods
class MyComponent extends Component {
  @On(SomeEvent)
  void _onEvent(_) {}
}
class OtherComponent extends MyComponent {
  @On(SomeEvent)
  void _onEvent(_) {}
}

// *** Annotations functionality by reflection ***
MethodMirror method = /* method annotated by @On(SomeEvent) */;
InstanceMirror component = reflect(new OtherComponent());
Function listener = component.getField(method.simpleName);
listener(event);
// `getField` is the only way how to access instance field
// and it always returns the most speficic method.
// -> Only the [OtherComponent#_onEvent] is called.

// *** Annotations functionality by transformers ***
class MyComponent extends Component {
  MyComponent(...) : ... {
    sendMessage(new SubscriptionRequest(SomeEvent))
      .stream.listen(_onEvent);
  }
  void _onEvent(_) {}
}
class OtherComponent extends MyComponent {
  OtherComponent(...) : ... {
    sendMessage(new SubscriptionRequest(SomeEvent))
      .stream.listen(_onEvent);
  }
  void _onEvent(_) {}
}
// The [OtherComponent#_onEvent] is called twice
// within [OtherComponent] instance.
```

The only solution for the reflection method is to force programmer to name methods uniquely. And this is not very convenient. If the transformers method is used, the transformer can easily rename the method, e.g. add unique suffix and the functionality will be correct. However, because transformer method it not currently used (see section 3.4.1), the issue remains unresolved.

Chapter 4

Conclusion

4.1 Framework overview

Outcome of this thesis is a novel web application framework with well analyzed base and architecture. The major results of the analysis are:

- Framework solves only issues that are mostly intact by existing web frameworks.
- The framework and components are designed to be fully modular and extensible.
- All the communication within an application flows through one central point.
- Architecture reduces programmer mistakes and contains internal security control mechanisms.
- It includes features to simplify application development and to create organized code.
- Architecture is compatible with many other architectural patterns like service-oriented architecture or event-driven architecture.
- Additional features important for web frameworks, like working with domain data, creating visual components or testing, are also analyzed.
- By its structure, the framework forces programmer to analyze components before they could be implemented.

The main concept is the central message bus that makes some advanced topics easy to implement. These are for example:

- Different types of communication — synchronous, asynchronous, point-to-point, ...
- Single interface for handling all these communication types.
- Precise security rules and restrictions of components scope.
- Loose component coupling and prevention of memory leaks.
- Interaction testing through events.
- Multi-level exceptions handling.

4.2 Implementation status

The framework is implemented in very basic, yet functional, version. The core classes and services are stable but not optimized for performance. Internal security mechanisms and framework extensions are in a draft phase, and are included only to demonstrate how they could look like if fully implemented.

4.3 Accomplished goals

At the beginning (section 1.2) seven goals were mentioned and they are met by the framework in the following extent:

Goal 1: Design universal framework. Framework architecture is very abstract, extensible, modular and general. No application-specific code is present and framework is therefore universal, usable for both simple and complex web applications.

Goal 2: Make it easy to use. The framework goes in some sense **against this goal**, because components are split into multiple classes and even a very simple application contains many components. At a first sight the created applications look complex and programmer would have to understand the underlying concepts to analyse the application quickly. This effect is partially mitigated by useful helpers (annotations), but the need to understand framework concepts before the applications can be programmed, still remains. The simplicity was partially sacrificed to keep other benefits the framework offers.

Goal 3: Prevent mistakes. Many typical programmer mistakes are eliminated by the framework architecture. It was demonstrated in the examples throughout this thesis.

Goal 6: Align with the base technology. The framework tries to meet the API and scenarios of the Dart language where possible.

Goals 4, 5 and 7 — Solve application-wide issues, Consider busines framework selection criteria, Include recommendations — were not addressed yet, or only partially, but should be covered with the first stable release.

4.4 Weak points and possible improvements

Some weak points from the technical point of view were already mentioned in the section 3.4. The following is a list of main drawbacks that have impact on the practical usage.

No visual components are present and web applications cannot be therefore developed instantly. Framework will have to contain adaptors to visual component libraries to be easily used and adopted by the Dart community.

Performance was not addressed yet, but is important for complex applications and framework would have to optimize the heavily used parts to be more efficient. On the other side, the browsers, Dart VM and Dart compiler performance improves heavily over time, so the optimizations do not have to be very deep.

Lack of tutorials that are important to fully understand framework concepts. Without them the framework could be used incorrectly. The framework code documentation should also become more helpful.

4.5 Future steps

The framework is implemented as a public Dart package called `dartbase`. The package preview is available through Dart Package Manager[1], at GitHub[2] and on http://www.dartbase.org homepage. The source code is released under very permissive open-source MIT license.

The next step is to collect feedback from the Dart community and with the insights decide which functionality should be included or improved in the first stable version `1.0.0`. Except for the functionality, also adequate tests set, documentation and tutorials should be ready with this version, so that the framework can be easily and immediately used.

[1] https://pub.dartlang.org/packages/dartbase
[2] https://github.com/miroslavraska/dartbase

Bibliography

[1] *Shadow DOM, W3C Working Draft* [online]. http://www.w3.org/TR/shadow-dom/, 2013-05-14 [cit. 2014-04-09].

[2] *Introduction to D-Bus* [online]. http://www.freedesktop.org/wiki/IntroductionToDBus/, 2013-07-14 [cit. 2014-04-10].

[3] *TodoMVC: Helping you select an MV* framework* [online]. http://todomvc.com/, 2013-08-06 [cit. 2014-03-05].

[4] *Dart: Frequently Asked Questions (FAQ)* [online]. https://www.dartlang.org/support/faq.html, 2013-12 [cit. 2014-05-01].

[5] *Harmony - ECMAScript Wiki* [online]. http://wiki.ecmascript.org/doku.php?id=harmony:harmony, 2014-01-30 [cit. 2014-04-09].

[6] *Dart API Reference (1.2)* [online]. https://api.dartlang.org/apidocs/channels/stable/, 2014-03-07 [cit. 2014-03-24].

[7] *The Dart Programming Language Specification (1.2)* [online]. https://www.dartlang.org/docs/spec/latest/dart-language-specification.html, 2014-03-07 [cit. 2014-03-24].

[8] *AMD* [online]. https://github.com/amdjs/amdjs-api/blob/master/AMD.md, 2014-03-17 [cit. 2014-04-09].

[9] *JavaScript Frameworks Market Share* [online]. https://wappalyzer.com/categories/javascript-frameworks, 2014-04-01 [cit. 2014-04-01].

[10] *Usage of JavaScript libraries for websites* [online]. http://w3techs.com/technologies/overview/javascript_library/all, 2014-04-01 [cit. 2014-04-01].

[11] *Dart Issues* [online]. https://code.google.com/p/dart/issues/list, 2014-04-03 [cit. 2014-04-03].

[12] *HTML 5.1 Nightly - A vocabulary and associated APIs for HTML and XHTML, Editor's Draft* [online]. http://www.w3.org/html/wg/drafts/html/master/single-page.html, 2014-04-09 [cit. 2014-04-09].

[13] *Collected Java Practices* [online]. http://www.javapractices.com/, 2014-05-01 [cit. 2014-05-01].

[14] *Dart VM and dart2js Performance* [online]. https://www.dartlang.org/performance/, 2014-05-10 [cit. 2014-05-10].

[15] Deepak Alur, Dan Malks, and John Crupi. *Core J2EE Patterns: Best Practices and Design Strategies (2nd Edition)*. Prentice Hall, 2003. ISBN 978-0131422469.

[16] Rob Ashton. *A comparison of various JS frameworks using cold hard data* [online]. http://codeofrob.com/entries/a-comparison-of-various-js-frameworks-using-cold-hard-data.html, 2013-05-28 [cit. 2014-04-01].

[17] Gerardo Canfora and Massimiliano Penta. Service-oriented architectures testing: A survey. *Software Engineering*, page 78–105, 2009. DOI 10.1007/978-3-540-95888-8_4.

[18] Mani K. Chandy. *Event-Driven Applications: Costs, Benefits and Design Approaches* [electronic]. Gartner Application Integration and Web Services Summit 2006, Caltech. http://infospheres.caltech.edu/sites/default/files/Event-Driven%20Applications%20-%20Costs,%20Benefits%20and%20Design%20Approaches.pdf, 2006.

[19] Doron Drusinsky and David Harel. On the power of bounded concurrency I: Finite automata. *Journal of the ACM*, 41(3):517–539, May 1994. DOI 10.1145/176584.176587.

[20] Mohamed E. Fayad, David S. Hamu, and Davide Brugali. Enterprise frameworks characteristics, criteria, and challenges. *Communications of the ACM*, 43(10):39–46, October 2000. DOI 10.1145/352183.352200.

[21] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002. ISBN 978-0321127426.

[22] Piero Fraternali, Sara Comai, Alessandro Bozzon, and Giovanni Toffetti Carughi. Engineering Rich Internet applications with a model-driven approach. *ACM Transactions on the Web*, 4(2):1–47, Apr 2010. DOI 10.1145/1734200.1734204.

[23] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994. ISBN 978-0201633610.

[24] Paulo Gonçalves. *Introduction to Project Management* [electronic]. Project Management Course, Università della Svizzera italiana, 2012.

[25] Kevlin Henney. *Five Considerations for Software Architects* [online]. http://www.infoq.com/presentations/Five-Considerations-for-Software-Architects, 2009-12-04 [cit. 2014-04-04].

[26] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. *Lecture Notes in Computer Science*, page 220–242, 1997. DOI 10.1007/bfb0053381.

[27] Seth Ladd. *Dart News & Updates: Ecma forms TC52 for Dart Standardization* [online]. http://news.dartlang.org/2013/12/ecma-forms-tc52-for-dart-standardization.html, 2013-12-13 [cit. 2014-05-10].

[28] Anne Lapkin and Deborah Weiss. *Ten Criteria for Selecting an Enterprise Architecture Framework* [online]. https://www.gartner.com/doc/838915/, 2008-12-15 [cit. 2014-04-01].

[29] Ari Lerner. *ng-book - The Complete Book on AngularJS*. Fullstack io, 2013. ISBN 978-0991344604.

[30] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008. ISBN 978-0132350884.

[31] Leesa Murray, David Carrington, and Paul Strooper. An Approach to Specifying Software Frameworks. In *Proceedings of the 27th Australasian Conference on Computer Science - Volume 26*, ACSC '04, pages 185–192, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[32] San Murugesan. Web Application Development: Challenges And The Role Of Web Engineering. In Gustavo Rossi, Oscar Pastor, Daniel Schwabe, and Luis Olsina, editors, *Web Engineering: Modelling and Implementing Web Applications*, Human-Computer Interaction Series, pages 7–32. Springer London, 2008. DOI 10.1007/978-1-84628-923-1_2.

[33] Bob Nystrom. *Idiomatic Dart* [online]. https://www.dartlang.org/articles/idiomatic-dart/, 2013-03 [cit. 2014-04-03].

[34] Mike P. Papazoglou and Willem-Jan Heuvel. Service oriented architectures: approaches, technologies and research issues. *The VLDB Journal*, 16(3):389–415, May 2007. DOI 10.1007/s00778-007-0044-3.

[35] Tero Piirainen. *Frameworkless JavaScript* [online]. https://moot.it/blog/technology/frameworkless-javascript.html, 2013-09-17 [cit. 2014-04-01].

[36] Alexy Shelest. *Model View Controller, Model View Presenter, and Model View ViewModel Design Patt* [online]. http://www.codeproject.com/Articles/42830/Model-View-Controller-Model-View-Presenter-and-Mod, 2009-10-03 [cit. 2014-03-05].

[37] Boris Staal. *2-Way Data Binding under the Microscope* [online]. http://staal.io/blog/2014/02/05/2-way-data-binding-under-the-microscope/, 2014-02-05 [cit. 2014-04-09].

[38] Nicholas C. Zakas. *Scalable JavaScript Application Architecture* [online]. http://www.slideshare.net/nzakas/scalable-javascript-application-architecture, 2009-09-07 [cit. 2014-03-05].

[39] Nicholas C. Zakas. *Maintainable JavaScript*. O'Reilly Media, 2012. ISBN 978-1449327682.

Index

Appendix A

Enclosed medium contents

Directories description

`framework` — The web application framework package.
`framework/dartdoc-viewer` — The documentation package.
`framework_polymer` — Base package for development with Polymer.dart UI library.
`polymer_bootstrap` — Package with Twitter Bootstrap theme for Polymer elements.
`polymer_elements` — Package with UI components used in the sample applications.
`sample_cms` — Sample content management system application.
`sample_email` — Sample email client application.
`sample_todomvc` — Sample TodoMVC application, described by the `tutorial`.
`todomvc_common` — Package with TodoMVC resources.
`tutorial` — Step-by-step example of application development on partially implemented TodoMVC application. Functional version is in the `sample_todomvc`.

Structure of the framework package

`lib/core` — Contains framework core — a small, encapsulated library containing:

- The message bus.
- Minimal skeleton of application component.
- Utils for components connection to the message bus.
- Mechanism for internal communication within components.

`lib/extensions` — Contains libraries and packages for extension of the framework core.

- `bindings` — Annotations for simpler programming of applications. They automatically create component associtations, subscribe to events, services, etc.
- `data` — Tools for working with application-wide domain entities.
- `helpers` — Helpers for working with the framework core.
- `hierarchy` — Placement of components into a logical hierarchy.
- `state_object` — State of component is placed into separate object. It could be then serialized, dynamically saved and recovered (e.g. from URL).

- `validation` — Validation of `@required` properties in the configuration, etc.

`lib/utils` — General extensions applicable to any Dart object.

How to run a sample application?

1. Install the Dart editor (Arch linux — the `dart-editor` package from AUR)
2. Open all projects with the `Open Existing Folder` option.
3. Download libraries for each project using `Tools -> Pub Get`.
4. Run `web/app.html` application with the `Run in Dartium` context menu option.
5. The Dartium browser with the application is opened.

How to generate and view the documentation?

The Dart way of publishing code documentation is with the `dartdoc-viewer` package, which is a single-page documentation viewer application. It works with the documentation generated in the JSON format. There is no static version of the generated documentation and the viewer can be either hosted or executed locally. To display the documentation locally follow these steps:

1. Install Dart SDK (it is part of the `dart-editor` package in Arch linux) and Git.
2. Add the installed tools into `PATH` (Arch linux — `/opt/dart-editor/dart-sdk/bin`).
3. Generate documentation in JSON format and download the newest version of `dartdoc-viewer` application: `cd framework/tool/ && ./generate_docs.sh`.
4. Documentation is placed into the `docs` directory in the JSON format.
5. Documentation viewer application is located in the `dartdoc-viewer` directory.
6. Disable code analysis for the documentation viewer application in the Dart editor (context menu `Don't analyze` option).
7. Run `framework/dartdoc-viewer/client/web/index.html` in the Dartium from the Dart editor.

Appendix B

List of compared frameworks and libraries

The complete list of frameworks and libraries that were compared during the framework analysis.

- **$$**
- **AJS**
- **Abaaso**
- **AccDc**
- **ActiveJS**
- **Agility**
- **Almond**
- **Ample SDK**
- **AmplifyJS**
- **AngularJS**
- **Archetype**
- **AriaTemplates**
- **Asana Luna**
- **Atom**
- **AtomJS**
- **AUI**
- **Axton**
- **Backbone.js**
- **Batman**
- **Bindows**
- **BoilerplateJS**
- **Bootstrap**
- **CanJS**
- **Cappuccino**
- **Chaplin**
- **Choco**
- **ChocolateChip**
- **Claypool**
- **Closure Tools**
- **CommonJS**
- **CorMVC**
- **Cujo**
- **CupQ**
- **D3JS**
- **Derby**
- **Dermis**
- **Descript**
- **DHTMLX**

- **Dijon**
- **Dojo Toolkit**
- **Duel**
- **Echo**
- **Eco**
- **Ember**
- **Ender**
- **Enyo**
- **EnvJS**
- **Epitome**
- **Eventd**
- **Eyeballs**
- **Fleegix**
- **Flight**
- **Funnyface**
- **GWT**
- **Glow**
- **GLOW**
- **Heisenberg**
- **HandlebarsJS**
- **Infusion**
- **Jamal**
- **JavaScriptMVC**
- **Jo**
- **Joosy**
- **jQuery**
- **jQuery UI**
- **jQueryMx**
- **Jquip**
- **jslibraryboilerplate**
- **June**
- **Kendo UI**
- **Kinetic**
- **Knockback**
- **Knockout**
- **LimeJS**
- **Lively Kernel**
- **Lodash**
- **Maria**
- **Marionette**
- **Mass**
- **Meteor**
- **Midori**
- **MochiKit**
- **Mojito**
- **ModuleJS**
- **Montage**
- **Moo.fx**
- **MooTools**
- **Mustache**
- **Nanoko**
- **Ojay**
- **Olives.js**
- **Opa**
- **PhantomJS**
- **Plastron**
- **Popcorn.js**
- **Prototype**
- **Protovis**
- **Processing.js**

- **PureMVC**
- **Qooxdoo**
- **QunitJS**
- **Qworum**
- **Raphael**
- **RAppid**
- **Rialto**
- **Rico**
- **RightJS**
- **Sammy**
- **scaleApp**
- **Script.aculo.us**
- **Sencha ExtJS**
- **Serenade**
- **SimpleJS**
- **Sizzlejs**
- **Smart**
- **SmartClient**
- **SnackJS**
- **SocketStream**
- **Soma.js**
- **Spine**
- **SproutCore**
- **Spry**
- **Stapes**
- **StealJS**
- **Strophe**
- **SweetDEV**
- **Terrific**
- **TheBeast**
- **Thorax**
- **TrimJunction**
- **Troop**
- **UIZE**
- **Underscore.js**
- **Vanilla**
- **Wakanda**
- **Wink Toolkit**
- **YUI**
- **Zepto**

Appendix C

Step-by-step example of application development

In this tutorial you will learn the basics of creating applications with the `dartbase` framework on the TodoMVC[1] application. It was originally designed to compare MV* frameworks but it could also nicely demonstrate features of this framework, although it is not MV* based. The application will use Polymer.dart[2] for the View functionality and Dartbase for Model- and Controller-related functionality.

C.1 The TodoMVC application

The application contains a list of Todos that can be dynamically added, removed, modified, completed or reopened. TodoMVC specification contains more features but this basic CRUD functionality is enough to demonstrate framework features, therefore the remaining features like filtering or routing are not addressed by this tutorial. At the end I will show a simple way how to persist entities in browser local storage.

Start with creating a new Dart web project with the basic structure. Add the `framework_polymer` package as a dependency — it is a version of the `dartbase` framework configured for working with Polymer.dart library. Include also the `todomvc_common` package that contains base TodoMVC skin and resources, to have them stored locally.

Listing C.1: `pubspec.yaml`

```yaml
name: todomvc
description: TodoMVC example with the dartbase framework.
dependencies:
  browser: any
  framework_polymer:
    path: ../framework_polymer
  todomvc_common:
    path: ../todomvc_common
```

The HTML template should include TodoMVC common files, so that all the TodoMVC applications look the same.

[1] http://todomvc.com/
[2] https://www.dartlang.org/polymer-dart/

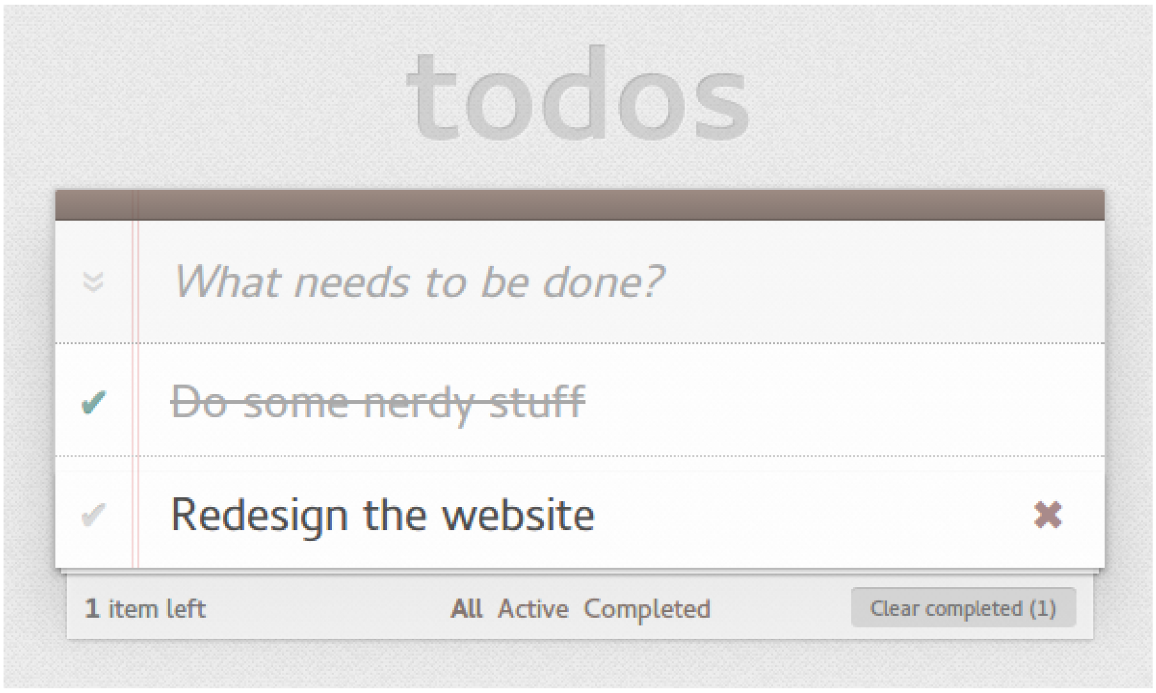


Figure C.1: TodoMVC Screenshot.

Listing C.2: app.html

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>TodoMVC</title>

    <!-- The TodoMVC skin -->
    <link rel="stylesheet" href="packages/todomvc_common/base.css
        ">
    <script src="packages/todomvc_common/base.js" defer></script>

    <!-- Application layout -->
    <link rel="stylesheet" href="app.css">

    <!-- Application initializer -->
    <script type="application/dart" src="app.dart"></script>
    <script src="packages/browser/dart.js"></script>
  </head>

  <body>
    <div id="main"></div>
  </body>
</html>
```

C.2 Create the application launcher

The purpose of the **app.dart** is only to create instance of the `Application` and launch it. Optionally any development tools and helpers should be initialized here.

Listing C.3: `app.dart`

```dart
library todomvc;

import 'dart:html' as dom;
import 'package:framework_polymer/framework.dart';

// Application initializers
part 'app/application.dart';

// The entry point --- should launch the application.
void main() {
  // Init the Polymer library.
  Zone polymerObservableZone = initPolymer();

  // Application is running without a global namespace.
  Application app = new TodoMvcApp(new TodoMvcAppConfig()
    ..viewport = dom.querySelector('#main')
    ..polymerZone = polymerObservableZone
  );
  app.run();
}
```

The `Application` implementation should initialize application components.

Listing C.4: `app/application.dart`

```dart
part of todomvc;

@PartOf(TodoMvcApp)
class TodoMvcAppConfig extends ApplicationConfig {
  // The element where application will be rendered.
  dom.HtmlElement viewport;
}

class TodoMvcApp extends Application {
  // Store properly typed config.
  final TodoMvcAppConfig config;
  TodoMvcApp(TodoMvcAppConfig config) :
    this.config = config,
    super(config ..bus = new MessageBus());

  void run() {}
}
```

C.3 Define domain data

Application works only with one domain class — `Todo`.

Listing C.5: `app/data/todo.dart`

```
part of todomvc;

class Todo extends DomainObject {
    int id;
    String title;
    bool completed;
}
```

C.4 Find *existing* visual components

One of the key ideas behind the framework is that visual components already exist in many libraries and can be easily reused, especially in such a simple applications like this one. Assume that we have found two general components that nicely fit into the TodoMVC concept.

1. An input element that reacts with custom DOM events when ENTER or ESC key is pressed. The element is available through `keyboard-events-input` tag or `KeyboardEventsInputElement` class and it fires the `enter-key` and `esc-key` events.

2. A simple list of items with the possibility to select items by a checkbox field, edit the item content or remove the item. The element is available through `dynamic-list` tag or `DynamicListElement` class and it fires the `item-edited` and `item-removed` events. It also contains few configuration options to change default class names in the generated DOM structure. The displayed data are passed through `DynamicListItem` model objects that contain 3 properties (`String id`, `String value` and `bool selected`). For usage within TodoMVC we will use the selection feature to mark Todos as completed.

C.5 Create visual components wrappers

Firstly, we need to import the reusable components to the application. Note that we use `ui` namespace for the reusable components to keep application namespace clean.

Listing C.6: `app.dart`

```
import 'lib/dynamic_list_element.dart' as ui;
import 'lib/keyboard_events_input_element.dart' as ui;
```

Now we can start creating the component wrappers that should add some behaviour to the reusable visual elements.

C.5.1 Todo creator field

The component should encapsulate the `ui.KeyboardEventsInput` and listen to its `enter-key` event. It should then create new `Todo` domain object and register it into the `DomainDataProvider`:

Listing C.7: `app/ui/todo_creator.dart`

```dart
part of todomvc;

@PartOf(TodoCreator)
class TodoCreatorConfig extends UIComponentConfig {
}

class TodoCreator extends UIComponent {
  // Constructor
  final TodoCreatorConfig config;
  TodoCreator(TodoCreatorConfig config) : this.config = config,
      super(config);

  // Property will be initialized from outside.
  ui.KeyboardEventsInputElement el;

  // Normally the [DomainDataProvider] should take care of this,
      but its implementations
  // is not yet complete, so IDs must be generated manually
      meanwhile.
  int _nextId = new DateTime.now().millisecondsSinceEpoch;

  // Subscription to DOM event through annotation.
  @On(DomEvent, name: 'enter-key')
  void _onDomEnterKey(DomEvent e) {
    // Create the domain object
    Todo todo = new Todo()
      ..id = _nextId++
      ..completed = false
      ..title = el.value.trim();

    // Register it into the [DomainDataProvider].
    dataRegister(todo);

    // clear value
    el.value = '';
  }
}
```

The framework tries to keep the implementation of interactions straightforward. The component therefore contains only obligatory constructor, one instance property with the DOM element and subscription to the DOM event.

C.5.2 List of Todos

The second component is the list of Todos that displays, modifies and removes domain entities. The `DynamicListElement` doesn't work with the domain data, so the `TodosGrid` class should map `Todo` domain objects into `DynamicListItem` model objects and vice versa. It should also react to domain data changes caused by other components in the application (e.g. it should refresh the list when new Todo is added).

Listing C.8: app/ui/todos_grid.dart

```dart
part of todomvc;

@PartOf(TodosGrid)
class TodosGridConfig extends UIComponentConfig {
}

class TodosGrid extends UIComponent {
  // Constructor
  final TodosGridConfig config;
  TodosGrid(TodosGridConfig config) : this.config = config, super
      (config);

  // Property will be initialized from outside.
  ui.DynamicListElement el;

  // Display existing Todos when component is created.
  @On(ComponentCreated)
  void _onCreated(_) => _refreshTodos();

  // Fetch the Todo based on the model object ID, then remove it.
  @On(DomEvent, name: 'item-removed')
  void _onDomItemRemoved(DomEvent e) {
    Todo todo = _getTodo(e.detail);
    dataRemove(todo);
  }

  // Fetch and update the Todo with the data from the model
      object.
  @On(DomEvent, name: 'item-edited')
  void _onDomItemEdited(DomEvent e) {
    Todo todo = _getTodo(e.detail);
    _applyModel(todo, e.detail);
    dataUpdate(todo);
  }

  // Subscribe to domain data changes and refresh the list (e.g.
      when Todo is added)
  @On(DomainDataChangeEvent)
  void _onDataChange(_) => _refreshTodos();

  // Fetches all the Todos, maps them to the model object and
      replaces element contents.
  void _refreshTodos() {
    // Get list of all Todos
    List<DomainObject> todos = dataQuery(Todo);

    // Update the displayed model data
    el.items
      ..clear()
      ..addAll(todos.map(_getModel));
  }


  // Creates model object from the Todo domain object.
  ui.DynamicListItem _getModel(Todo todo) {
    ui.DynamicListItem item = new ui.DynamicListItem(todo.id.
        toString())
      ..selected = todo.completed
      ..value = todo.title;
    return item;
  }

  // Applies model object changes to the Todo domain object.
  void _applyModel(Todo todo, ui.DynamicListItem item) {
    todo.completed = item.selected;
    todo.title = item.value;
  }

  // Finds Todo by the ID from the model object.
  Todo _getTodo(ui.DynamicListItem item) {
    int id = int.parse(item.id);
    return dataFindFirst(Todo, (Todo t) => t.id == id);
  }
}
```

And we are done with the interaction logic. The basic CRUD functionality is implemented in 100 lines of code. The components are event-driven and modular.

C.6 Create application layout

Now it is time to connect our reusable elements and two application components together with a new polymer element, that would serve as an application layout.

Listing C.9: `app/layout/main_layout.html`

```
<link rel="import" href="../../lib/dynamic_list_element.html">
<link rel="import" href="../../lib/keyboard_events_input_element.
    html">

<polymer-element name="main-layout">
  <template>

    <section class="todoapp">
      <header class="header">
        <h1 class="h1">todos</h1>

        <!-- create instance of the [KeyboardEventsInputElement]
            -->
        <input is="keyboard-events-input" id="todoCreator" class=
            "new-todo"
            placeholder="What needs to be done?" autofocus />
      </header>

      <section class="main">
        <!-- create instance of the [DynamicListElement] -->
        <dynamic-list id="todosGrid" listClass="todo-list"
          selectedClass="completed" checkboxClass="toggle"></
              dynamic-list>
      </section>
    </section>

    <footer class="info">
        <p>Double-click to edit a todo</p>
        <p>Created by <a href="http://todomvc.com">you</a></p>
        <p>Part of <a href="http://todomvc.com">TodoMVC</a></p>
    </footer>

  </template>
  <script type="application/dart" src="main_layout.dart">
  </script>
</polymer-element>
```

Listing C.10: `app/layout/main_layout.dart`

```
library main_layout;

import 'package:polymer/polymer.dart';
import 'dart:html';

@CustomTag('main-layout')
class MainLayoutElement extends PolymerElement {

  HtmlElement todoCreator;
  HtmlElement todosGrid;

  MainLayoutElement.created() : super.created();

  bool get applyAuthorStyles => true;

  @override
  void enteredView() {
    super.enteredView();

    todoCreator = $['todoCreator'];
    todosGrid = $['todosGrid'];
  }
}
```

Don't forget to import the custom HTML element.

Listing C.11: `app.html`

```
<!-- Application layout -->
<link rel="import" href="app/layout/main_layout.html">
```

Now we have to create the layout element instance, append it to the DOM and initialize instance variables of application components:

Listing C.12: `app/application.dart`

```
class TodoMvcApp extends Application {
  // ...

  void run() {
    // Create application components
    TodoCreator creator = new TodoCreator(new TodoCreatorConfig()
      ..app = this
    );
    TodosGrid grid = new TodosGrid(new TodosGridConfig()
      ..app = this
    );

    // Render the application
    MainLayoutElement layout = createPolymerElement('main-layout'
       );
    this.config.viewport.append(layout);
    creator.el = layout.todoCreator;
    grid.el = layout.todosGrid;
  }
}
```

C.7 Create application managers

The last step is to create application managers required by the framework — `StateStorage`, `StateManager` and `HierarchyProvider`. They are not used in this application, but since we have based our application on the extended framework version, they have to be included.

We are using one class from the framework core — the `DomainDataProvider`, which comes in two versions. The `MemoryDomainDataProvider` that does not persist entities and the `LocalStorageDomainDataProvider` that persists entities in browser local storage. We will use the second one to demonstrate how simple is to store data.

Listing C.13: `app/application.dart`

```
class TodoMvcApp extends Application {
  // ...

  void run() {
    // Core Managers
    UrlStateStorage stateStorage = new UrlStateStorage(new
       UrlStateStorageConfig()
      ..app = this
    );
    StateManager state = new StateManager(new StateManagerConfig
       ()
      ..app = this
    );
    HierarchyProvider hierarchy = new HierarchyProvider(new
       HierarchyProviderConfig()
      ..app = this
    );


    // Persistent data storage
    DomainDataProvider provider = new
        LocalStorageDomainDataProvider(
      new LocalStorageDomainDataProviderConfig() ..app = this
    );

    // ...
  }
}
```

C.8 Include development tools

C.8.1 Logger

If logger is set up to the `Level.ALL`, it will show internal messages from the framework core that might be useful for debugging. At `Level.INFO` it shows only caught and resolved exceptions, plus all the severe issues and messages. If logger is not present, **no warnings or errors are shown**, so it is highly recommended that a logger is present in every application.

Listing C.14: `app.dart`

```
void main() {
  // ...

  // Init logging
  Logger.root.level = Level.INFO;
  Logger.root.onRecord.listen((LogRecord rec) {
    print('${rec.loggerName}: ${rec.message}');
  });
}
```

C.8.2 Message bus watcher

Is a GUI component for watching all the flow on the message bus. It could be included in the following way:

Listing C.15: `application.dart`

```
class TodoMvcApp extends Application {
  // ...

  void run() {
    // ...

    // Debugging
    MessageBusWatcher watcher = new MessageBusWatcher(new
        MessageBusWatcherConfig()
      ..app = this
    );

    // ...
  }
}
```

Listing C.16: `app.html`

```
<!-- Devel tools -->
<link rel="import" href="packages/framework_polymer/devel/
    message_bus_watcher_element.html">
```

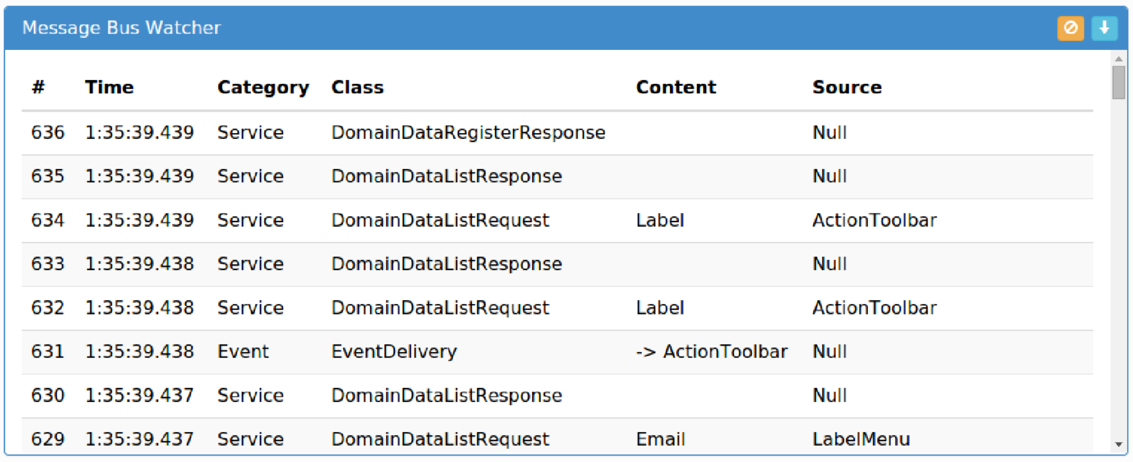
Listing C.17: `app/layout/main_layout.dart`

```
@CustomTag('main-layout')
class MainLayoutElement extends PolymerElement {
  // ...
  HtmlElement watcher;

  @override
  void enteredView() {
    // ...
    $['watcher'].append(watcher);
  }
}
```

Listing C.18: `app/layout/main_layout.html`

```
<polymer-element name="main-layout">
  <template>
    <!-- ... -->
    <div id="watcher"></div>
  </template>
</polymer-element>
```



Message Bus Watcher

#	Time	Category	Class	Content	Source
636	1:35:39.439	Service	DomainDataRegisterResponse		Null
635	1:35:39.439	Service	DomainDataListResponse		Null
634	1:35:39.439	Service	DomainDataListRequest	Label	ActionToolbar
633	1:35:39.438	Service	DomainDataListResponse		Null
632	1:35:39.438	Service	DomainDataListRequest	Label	ActionToolbar
631	1:35:39.438	Event	EventDelivery	-> ActionToolbar	Null
630	1:35:39.437	Service	DomainDataListResponse		Null
629	1:35:39.437	Service	DomainDataListRequest	Email	LabelMenu

Figure C.2: Message bus watcher.