



**FACULTY
OF MATHEMATICS
AND PHYSICS**
Charles University

BACHELOR THESIS

Michal Bureš

Startpage for TV application

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: Doc. RNDr. Tomáš Bureš, PhD.

Study programme: Computer Science

Study branch: Programming and Software Systems

Prague 2017

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In date

signature of the author

Title: Startpage for TV application

Author: Michal Bureš

Department: Department of Distributed and Dependable Systems

Supervisor: Doc. RNDr. Tomáš Bureš, PhD., Department of Distributed and Dependable Systems

Abstract: In this thesis we implement configurable set-top box TV application in context of an embedded web browser. Work is executed in accord with strict customer requirements and fully integrated into large IPTV platform. It takes form of a complex start-up menu, whose first section is an interactive tile grid with tile-scaled video player. Other sections contain smaller applications or sub-sections. We build upon modules that transform data sources into tiles. They are then projected to horizontal carousels according to dynamic configuration. A well defined interface to integrate new modules is provided. Almost twenty different modules are ready for immediate use. They provide various parametrized tiles such as live programmes that can be directly started or recorded. Our user interface layer, using React and Redux libraries, leverages the single page application paradigm. For predictability, any modification of application state is made by emitting actions handled solely by pure functions. Set-top box performance issues forced us to implement an immutable state optimization that cut the average render time of our React components by more than a half.

Keywords: Javascript, Set-top-box, Web Development, React, Redux

Název práce: Startpage pro TV aplikaci

Autor: Michal Bureš

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Doc. RNDr. Tomáš Bureš, PhD., Katedra distribuovaných a spolehlivých systémů

Abstrakt: V naší práci implementujeme konfigurovatelnou TV aplikaci pro set-top box zařízení, v kontextu zabudovaného webového prohlížeče. Práce je vedena dle zákaznického zadání a plně integrovaná do velké IPTV platformy. Má formu komplexního výchozího menu, jehož první sekcí je interaktivní dlaždicová struktura obsahující pohyblivý video přehrávač. Ostatní sekce obsahují menší aplikace, nebo podsekce menu. Základem jsou moduly, které transformují zdroje dat na dlaždice. Ty jsou pak projektovány na horizontální karusely dle dynamické konfigurace. Navrhli jsme vhodně definované rozhraní k integrování nových modulů. Necelých dvacet jich je připraveno k okamžitému použití. Poskytují různé druhy parametrizovatelných dlaždic, například živé programy, s možností je přímo pustit či nahrát. Naše vrstva uživatelského rozhraní využívá knihoven React a Redux a je navržena jako jednostránková aplikace. Pro předvídatelnost, jakákoli změna stavu aplikace je řešena prostřednictvím vytvoření akce, která je zpracována pouze pure funkcemi. Nedostatečný výkon set-top boxů nás přivedl k implementaci pomocí immutable stavu aplikace, po kterých se průměrná doba renderování našich React komponent zkrátila o více než polovinu.

Klíčová slova: Javascript, Set-top-box, Webové technologie, React, Redux

I would like to express my sincere gratitude to my supervisor for his precise comments and efficient communication. A special mention goes to my outstanding colleagues, as they provided me with invaluable assistance and advice.

Contents

1	Introduction	3
1.1	Goals	4
1.2	Outline	4
2	Requirements	5
2.1	Administration requirements	5
2.1.1	Basic platform structure	5
2.1.2	Configuration requirements	5
2.2	Functional requirements	6
2.3	Set-top box development	8
3	Analysis	9
3.1	Single-page applications	9
3.2	Portal structure	10
3.2.1	Applets	10
3.2.2	Current framework	10
3.2.3	Problems	10
3.3	Libraries	11
3.3.1	Framework comparison	11
3.3.2	Choosing React & Redux	11
4	Technology background	13
4.1	Redux	13
4.1.1	Nomenclature	13
4.1.2	Immutability	14
4.1.3	Functional Programming	15
4.1.4	Sagas	15
4.2	React	15
4.2.1	Functional Components	15
4.2.2	Virtual DOM	15
4.2.3	Reconcillation	16
4.2.4	Optimizations	16
4.3	Webpack	17
4.3.1	Entry	17
4.3.2	Output	17
4.3.3	Loaders	17
4.3.4	Plugins	17
4.4	EcmaScript 6	18
4.4.1	Modules	18
4.4.2	Generator functions	18
5	Implementation	19
5.1	Architecture	19
5.1.1	General structure	19
5.1.2	Files structure	20

5.2	Configuration service	22
5.3	Data service	23
5.3.1	Sources manager	23
5.3.2	Tiles cache	23
5.3.3	Managers and applications	23
5.4	User interface layer	24
5.4.1	Reducers	25
5.4.2	Middleware	26
5.5	Menu applications	29
5.6	Tile sources interface	30
5.6.1	Contract	30
5.6.2	Producing tiles	31
6	Evaluation	33
6.1	Browser Profiling	33
6.2	React & Redux developer tools	34
6.3	Rendering Optimality	35
6.4	Measuring the effect	36
6.5	Player in picture	37
7	Conclusion	38
7.1	Implementation assessment	38
7.2	Future work	38
	Bibliography	39
A	Documentation	40
A.1	Basic configuration	40
A.2	Grid configuration	41
A.2.1	Row definition	41
A.3	Available tile types	42
A.3.1	Types producing multiple tiles	42
A.3.2	Types producing a single tile	46
	List of Figures	59
	List of Tables	60
	List of Abbreviations	61
	Attachments	62

1. Introduction

Nowadays, television services are an ever present standard. In 2013, 79% of all households owned a TV set [1]. Operators, striving for customers, provide services such as television over IP, non-linear (archived) TV, personalized video content or custom applications.

In this thesis we extend an existing IPTV portal, running on set-top box (STB) devices, with a modern startpage. The project has to be implemented in line with given customer requirements. The old textual menu is to be replaced by a complex application that will serve as a multi-functional home screen with live data. In the following text we refer to it as the Startpage (SP) application or simply as application.

Our work will be done in context of a complex platform suited for large operators to provide both IPTV and over-the-top (OTT) video streaming. We will frequently talk about the multi-functional system running on STB device providing not only video streaming but many other useful features. Throughout the text we refer to it as STB portal. Another frequently mentioned component of the platform is the application server that serves as a source of content and meta-data for large groups of end devices. Besides STBs the server provides data for mobile devices. This way users can enjoy basic TV services on their mobile devices.

Modern TV sets contain various applications based on different systems. Their functionality is diverse and often focuses on non-live content. Unfortunately, they often lack non-linear TV services and are outlived by the hardware. A possible solution is using a unified platform on a set-top box device that can be connected to any modern TV. It can provide aggregation of live and non-live content which is an advantage over standalone solutions. It results in a stable solution with consistent experience across devices. The STBs are relatively cheap and can easily be replaced by the provider.

The set-top box devices have other capabilities besides turning the source signal to video output. In most cases there is a Linux system running that enables further utilization of the hardware. They come with APIs that enable using functionality of the STB connected with the video output. In this way applications can be created with much more user-friendly interfaces.

The STB portal is a javascript-driven web application running in an embedded web browser, mostly Webkit or Mozilla. The video is streamed into some special element behind the browser DOM. Thus we are working in the context of web browsers, we can use the same technologies and leverage the skills of web developers. However, we have to keep in mind that we work with weaker hardware, the devices contain older version browsers and our applications may have to run continuously for many days.

1.1 Goals

The goal of this thesis is a deep rework of the central page of the STB portal. We take customer requirements and turn them into a production ready application.

The SP application is designed to be a complex start-up menu. First section will contain a grid-like structure of horizontal carousels. They will be filled with interactive tiles providing shortcuts to frequent actions in the STB portal. Other sections will integrate existing applications or serve as menu sub-sections.

We will focus on satisfying the requirements, while implementing an efficient and well maintainable application. Performance is a non-negligible parameter of our implementation. We have to make sure that the STB experience is satisfactory at least on the targeted devices. In Chapter 2, we describe the requirements in greater detail.

1.2 Outline

The thesis begins with introducing our goals and presenting the outline. We also provide some contextual background for set-top box development.

Chapter 2 contains the fixed requirements on the application. At first, we analyze them from the administrator's point of view. Secondly, we focus on most important functional requirements.

Chapter 3 presents our analysis not only from the perspective of the application alone but also as a future part of the ecosystem. The current framework of STB portal is analyzed. Taking into account its problems we discuss the choice of technologies and libraries used in our implementation.

Chapter 4 presents basic concepts of the libraries we decided to use. It focuses mainly on the features required to understand the implementation. We present the functionality of used libraries that was helpful and made strong impact on the structure and paradigms of the final implementation.

In Chapter 5 we provide an overview of the implementation and our architectural decisions. Its contents serve as a documentation to any contributor. We explain the structure of the project and the basic flow of the program. It also contains an overview of the interface for tile source modules.

Chapter 6 shortly evaluates our implementation and discusses performance testing. We focus mainly on the effect of the optimizations we made.

In Conclusion we evaluate the final implementation from the perspective of both hard and soft requirements. Finally, we discuss the space for possible extensions.

2. Requirements

In this chapter we present the requirements and constraints for our application. Most of them are provided by the customer. The implementation is based mainly on a design manual that describes the structure and behaviour of the page under varying circumstances. However, the manual is not exhaustive and many features had to be resolved ad-hoc by mutual agreement. Second important source of information is the list of functionality the tile sources should provide.

2.1 Administration requirements

This section discusses the basic integration and configuration. It enumerates the necessary levels of configuration that have to be available in order to fit the required use cases.

2.1.1 Basic platform structure

SP application has to communicate mainly with two systems.

The STB portal provides access to local information and to services that can be used to get additional data from the server. Our application has to enable easy access to the other sections of the portal. It has application has to seem like an integral part of the STB portal.

Secondly, the application server provides data mainly about users, movies, channels or programmes via REST API. Many of the tiles should be constructed using new API possibilities, preferably via `/subscription/search/search`. This method uses Elastic search on the server so it is handled more efficiently. This should enable the server to respond to more requests.

2.1.2 Configuration requirements

The application has to be highly customizable in three different aspects.

Runtime configuration by JSON

The requirements on the structure of the large grid can vary across customers and change over time. This means there has to be an option to change the configuration easily. A prefer options is a JSON file adhering some definition that can be configured by administrator and provided by the application server. Another level of configuration is the menu. It can be customer specific and has to reflect current capabilities (STB, off-line mode).

Easy addition of other tile modules

It is necessary that extending the functionality by new tile types is easy. Writing only a simple module in a single file should suffice. The interface has to be stable and well defined.

Different content for each user

Each user has different content available. He has a specific programme in

his player, different active channel list, recordings, activity or recommendations. This has to be ensured by querying the server for personalized content and observing the state of the whole application.

2.2 Functional requirements

It is important that the user interface simplifies the access to the most frequent actions. This is ensured by concentrating the most interesting content to the SP and presenting it in a visual manner. Thus just after starting the TV user has many suggestions how to entertain himself.

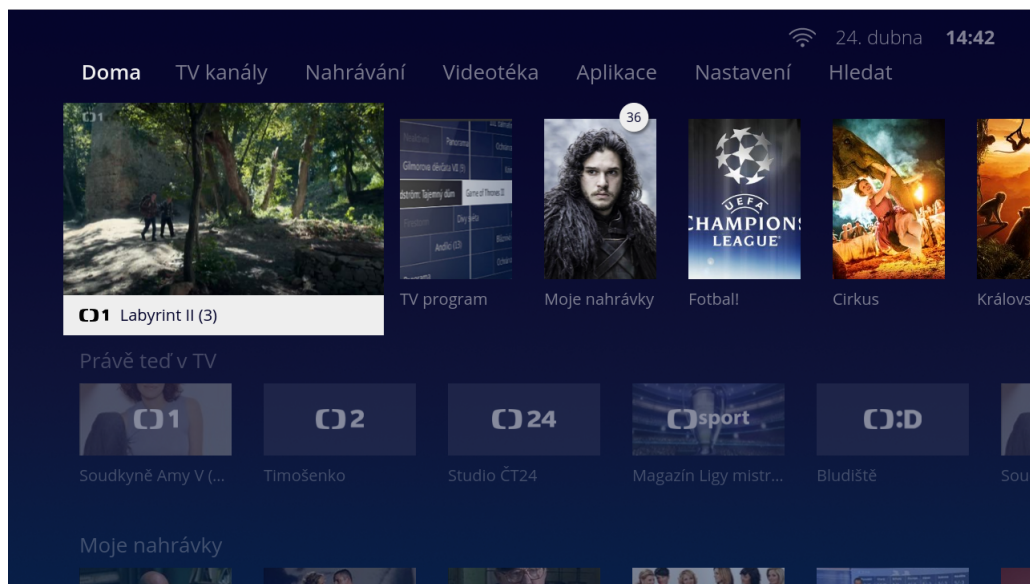


Figure 2.1: Application visual – screenshot

Grid menu with carousel

The tiles are organized in a two dimensional array with carousel functionality. Each tile is at least an image with an action that is invoked when selecting that tile. There can be more actions bound to a single tile invoked on another key on the remote controller (RC). For instance red key can start recording of a TV programme and ok key can open its detail.

The individual rows can be wider than the visible space on the page. This should be solved by sliding the row so that the selected tile is always completely in the specified visible area. When a tile out of the area is to be selected the whole row has to be slided to move the tile to the visible area.

Player in picture

It is important not to hide the player from the user when in SP. SP, unlike the former main menu, is designed to be a startup screen. It enables booting directly to the SP, not to the player. The solution to be used in SP is embedding the player in tile. This is done by moving the player to the higher layers of the DOM.

This is the player in picture functionality provided by most STB APIs. It has to be determined how to have this done, not all devices provide a direct solution.

Embedded menu

The application has to contain not only the grid based interface but also textual menu to provide access to less used applications and to maintain the former functionality. It has a new simple design and each menu section contains a custom banner that can be configured remotely. The right banner has to be available immediately after section change.

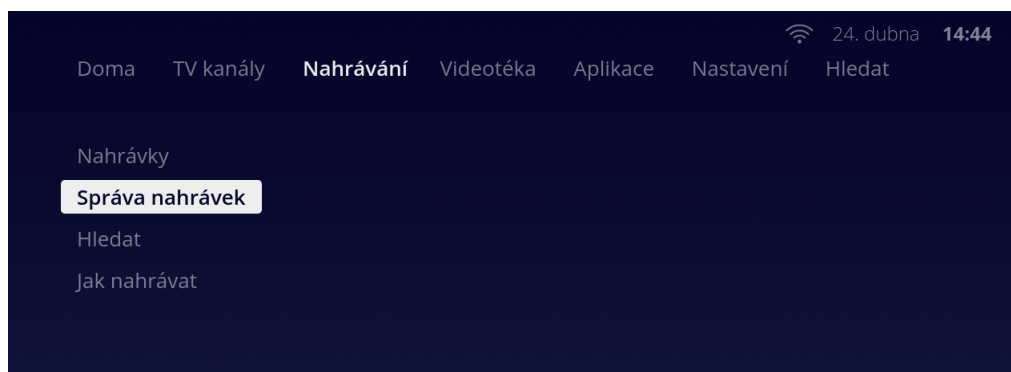


Figure 2.2: Embedded menu – screenshot

Data source modules

The cornerstone of this application have to be the modules that specify the behaviour of different tiles. The application has to support a dozen of tile types configured by the JSON file. So we need a reliable system how to write them, and a clear specification of the features they have to satisfy.

An important aspect of the tile functionality is that they have to react to any events in their data sources. When a new recording is completed or programme on a channel changes it has to be pushed to be observed by the SP application. The corresponding tiles should change or new tiles should appear as a result.

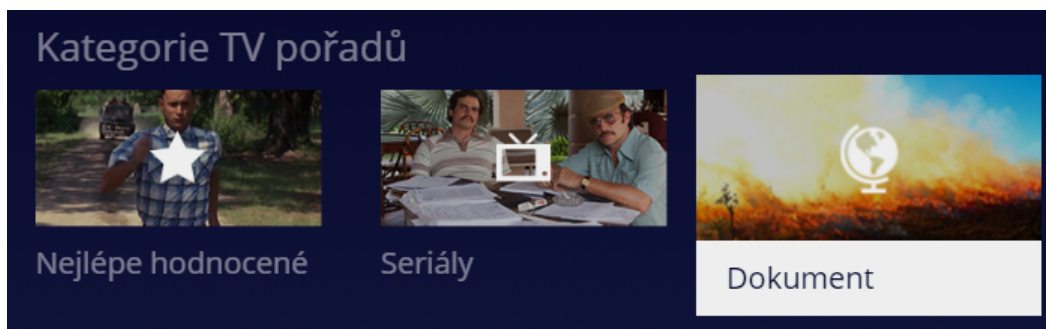


Figure 2.3: Tiles – screenshot

Reloading the application with different configuration should be available at runtime without any other visible actions. It may be required to react to limited access to resources such as in offline regime.

As there are almost twenty modules required we encourage the reader to consult Appendix A that provides list of all sources that were part of the requirements.

Menu containing applets

Textual menu that can navigate the user to functionality unavailable directly from the startpage also has to embed focusable applications. They have to be a natural part of the page and navigation between them should not bring confusion. An interface to control these applications has to be designed. Two such applications are a part of our implementation. Primarily, it is the grid application in the first menu section. We also have to integrate an already implemented search application to fit our needs and use it as a part of the menu.

2.3 Set-top box development

To give the reader more precise image of the devices we list the configuration of the STB model that is the primary target of SP application. It can be found in Listing 1.

```
EVO4-T
CPU:
ARM A9 1700 DMIPS Entropic TSC131 Processor
Memory:
512MB 1033MHz DDR3 (2 x 256MB)
512MB NAND Flash
Resolutions:
576i/576p/720p/1080i/1080p
MPEG-4 Part 10 AVC HD/ H.264 HP@L4, MP@ L3
```

Listing 1: Target STB model configuration

Development of applications for set-top box devices has one basic aspect. We always have to work with some API provided by the STB vendor. The application itself can be supported by many various technologies. Startpage is targeting web browser that are available in those devices.

Many of our problems can be solved from the perspective of web development. However, we have to take special care about performance and support of new standards. Naturally, the STB devices lag behind personal computers both in adoption of new browsers and the computational power available. We also have to care about memory usage, as the memory of this devices can be scarce.

Another feature is that such an application is often long-running. We cannot refresh the page as easily as in traditional scenarios. This means that we have to take extra care of memory leaks. They can be disastrous as they can cause gradual slow-down of the application and even lead to forced restarts of the device.

During our work we have to be aware of these risks and take necessary precautions to avoid them. We are creating web application in environment where we cannot afford unnecessary operations or excessive memory consumption.

3. Analysis

This chapter is devoted to the pre-implementation stage. It uncovers the core decisions and presents the reasoning they were based upon. We explain our architectural choices and to reveal the reasons why we use the chosen tools and libraries.

Our project has a set of important features. It is a web page, but classical URL routing does not make any good sense in our context. We need to avoid any visible reloads. Smooth transitions between sections are necessary. Persistent application state is also a desired aspect. When traversing between the screens we need to have everything prepared and manage the state efficiently.

The server load caused by the client should be minimal. We have to avoid unnecessary requests. Any inefficiency in this area will result in superfluous requests from thousands of devices.

In this situation we decided to implement the Startpage as a single-page application. We show why this architecture is the best fit for our problem.

3.1 Single-page applications

In Mesbah and Van Deursen [2] one of the early papers about moving to SPA structure, we can find a reasonable list of defining features of SPAs.

- Although representing a whole application, it is divided into many components. The components interact with each other and can be composed.
- The application is typically not reloaded during a single session. However the content can be changed dynamically.
- SPAs are heavy on user interaction and is responsible for handling all important inputs.

Developers are recommended to consult books, that provide an overview of the essential topics, before they undergo the adventure of writing a SPA. During implementation of our application we also consulted mainly Scott [3].

Features and benefits of SPAs are discussed in Mikowski and Powell [4]. It is argued that javascript SPAs are the best choice for modern web development. They list several reasons.

Primarily, they mention much more engaging user experience. This can be caused by the similarity to desktop application which are more responsive. They also write about application wide state. The advantage is observed mainly in systems where the individual pages share a lot of information. Often the data are more logically bound with the session than with individual page. In these cases the data can then be reused in contrast to the usual stateless pages. An important aspect is the javascript environment which nowadays provides all the necessary tools to write complex SPAs. This only supports our decision to use the SPA architecture.

3.2 Portal structure

The STB portal itself is a large SPA. The reasons are similar to those explained in our case. However, the portal contains many small application that have to be managed by a custom system.

3.2.1 Applets

The modules representing separate UI entities in portal are frequently called applets. They are managed by a custom window manager that can start any applet, force it to become focused, unfocused, selected, unselected or closed. Applets have to implement this interface to remain in a consistent state and to free resources properly across any state transitions.

3.2.2 Current framework

Behind the applet abstraction an UI framework is used to express the logic and presentation of individual applets. It is a combination of `doT.js` templating system, `rivets.js` data-binding library and other custom extensions that emerged as a fast solution to urgent needs. It is build on the model, view, view-model basis but throughout the time many deficiencies became apparent.

Applet became the basic building block. As this cannot be sufficient applets can dynamically use components that can implement part of their functionality. The applet mainly consists of all three parts of the model, view, view-model (MVVM) architecture.

Applet has to explicitly specify usage of a component both in view-template and in view-model. An instance of an event object that can pass events is created in the applet's view-model and passed to the component's controller. After the component template is loaded it is bound with that controller.

The controller is a special view-model whose only interface is the events extended object injected into its constructor. Then the applet can control the component with triggered signals through that object.

Alternatively, the component can trigger events in the other direction. That makes it sometimes hard to reuse components as they have to be quite universal and often do not suit new needs precisely. The components are written to serve to the first applet. When we want use them in another applet, the applet often has to be modified to suit to the component.

3.2.3 Problems

The problems we encounter frequently are:

- The layered structure is hard to maintain and the reasoning is often chaotic. Lot of communication is based on weakly typed signals that are triggered. They often have to bubble through three or four layers without any added functionality. Any changes that have to be made in those long chains of function calls are painful and can easily break other functionality.
- The model and view-model are often mixed together chaotically. There are different practices used across various applets.

- During the long existence of the project, mistakes once made propagated to newly written modules. Code was often copied and modified to solve the current problem instead of being written to model the situation. This often led to inefficient and complicated solutions.
- The system based on triggers can be bent to do almost anything but the solutions tend to be over-complicated and certainly are not transparent.

Accounting for the problems with the current framework we decided to find a more modern solution that can help us eliminate old problems in new solutions. During analysis we identified the best layer where to integrate new framework. The applet interface seemed to be the thinnest and well-defined border, that was followed without exception throughout the portal.

3.3 Libraries

Community around Javascript is more vibrant than it ever was. There are currently many libraries that can prove useful when used in our solution. Many useful comparisons of currently used frameworks can be found on the web [5, 6]. We compared currently most popular frameworks in how they suit our needs.

3.3.1 Framework comparison

We decided to explore `Angular2`, `Ember`, `React`, `Vue` and `Meteor`. In the case of `Angular2` the broad options it offered were impressive. However, the syntax has gone far from the natural javascript and seemed a little messy. `Ember` is discussed to have a steep learning curve and seems to bring a lot of complexity. Both `Vue` and `React` seemed to be quite accessible with their focus on declarative components. The former seemed to work lot more with bindings while the latter viewed components more as functions of data. Unfortunately, `Meteor` showed to be too large-scale for our context, more suited to complete solutions.

In Figure 3.1 we present a comparison of the scale the frameworks declare to operate on. Partial solutions exist so using the complete stack is not strictly necessary.

Ultimately we decided to use `React`. It is probably the most popular library, a common skill among javascript programmers and we like the declarative simplicity. During the process we also became acquainted with `Redux`. After seeing an example application and reading the documentation we decided to try it in our project.

3.3.2 Choosing React & Redux

Nevertheless, the primary reason why we decided to use `React` is that we can use it fully even in setups where rewriting the whole applet is not possible. We can just replace the template with react component and adjust the direct dependencies. In places where we are implementing all of the layers, we can use other libraries, possibly different to care about the logic. In our project we decided to use `Redux` as it can help us tame our large application state and feed the `React` components consistently.

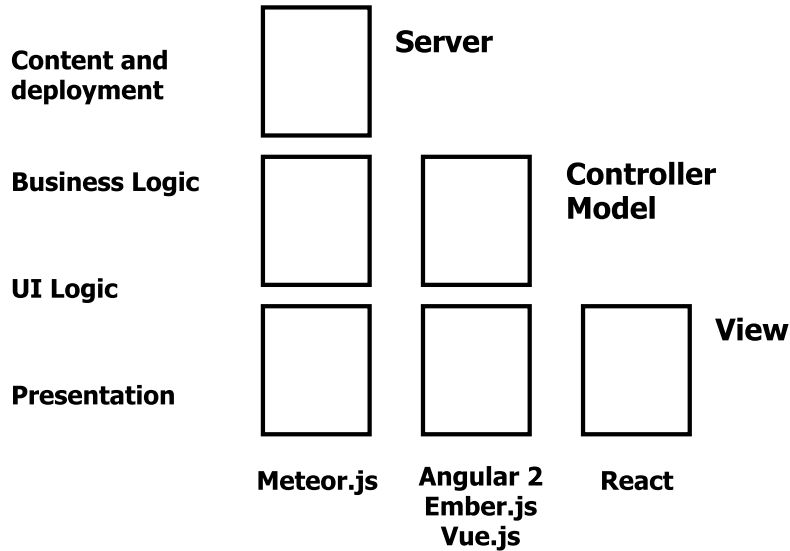


Figure 3.1: Frameworks scale comparison

Besides our personal preference we try to list the strengths of React & Redux according to our opinion:

Fast integration

React and Redux solve just a part of the problem without forcing us into unnecessary complexities, they seemed to be well suited for our task.

Focuses on predictability

Redux greatly fitted in our requirement for tracking the state changes. It not only offers a solid solution but is based on beautiful functional programming principles.

React is simple and popular

React has a simple system and an uncomplicated syntax that is in line with classical javascript. It does not bring a complicated DSL and has a decent learning curve.

Strong synergy

Redux was designed with React in mind. There are recommendations how to combine them to maximize the synergy. There are also many successful projects built with React & Redux.

Alternative to bindings

Data bindings are powerful. Unfortunately, they often result in chaotic that is split between templates and models.

4. Technology background

In this chapter we explain the basic concepts of libraries we used. We focus on their subset that is necessary to understand the next chapter dealing with implementation. We focus primarily on the features we used.

4.1 Redux

Most of the information presented here is based on our experience with the library mixed with information from the Redux documentation [7]. It states three principles of Redux that capture the gist of its usage.

Single source of truth

The whole state is stored in a tree-like structure using single javascript object. It enables easy implementation of undo & redo behaviour or state serialization.

Read-only state

The state cannot be modified in any other way than by emitting an action. Action is a plain object describing what happened. As a result a new state is created on top of the old one.

Mutations with pure functions

The only way to replace the current state with a new one is to handle and action in a pure function called reducer which can be composed of other reducers arbitrarily.

Redux is built upon the Flux architecture. It applies functional programming principles taken mostly from Elm to make the state modification predictable.

4.1.1 Nomenclature

Here we present basic terms we encounter in Redux applications.

Store

The store is a central object in every Redux application. In contrast to our libraries, Redux API is minimal. This can be exhibited even on the store object, illustrated in Listing 2.

Actions

Objects that represent actions are the only information that flows to the store. They must have the `type` property which is used to decide what how the action should be managed. It can have any amount of additional parameters. The actions can be stored to replay the behaviour at later stage or to be used for serializing the state history.

```

// Redux store API
// get copy of state object
getState()
// calls middleware and reducers potentially modifying state
dispatch(action)
// subscribe to changes of state
subscribe(listener)

```

Listing 2: Redux store API

Reducers

Reducers are pure functions that modify the state. They receive an action and a state (or sub-state) as parameters and return a new state instance. This is the main essence of Redux.

```

// (state, action) => newState
function selectableReducer(previousState, action) {
  switch (action.type) {
    case SELECT:
      return previousState.merge({ selectedItem: action.index });
    case DESELECT:
      return previousState.merge({ selectedItem: null });
  }
}

```

Listing 3: Simple reducer

Middleware

Middleware in the context of Redux is a chain of composed functions. They get access to the action before reducers do. They do not modify anything in store, but can perform side-effects or dispatch new actions both synchronously and asynchronously.

4.1.2 Immutability

Immutable object, once created, cannot be modified. It retains its shape through its lifetime. Redux manages its state as immutable. Unfortunately, the immutability is only a convention so it may be useful to use some library that can enforce it.

Javascript objects are not immutable. But if we do not mutate it or use some immutable data structure we can use it to improve performance. The ability to do a reference comparison on objects to determine if they changed can prevent unnecessary rendering.

4.1.3 Functional Programming

It may now seem clear that Redux is inspired by functional programming. It is heavily focused on function composition. Reducers are pure functions and return new instances. This can provide a clean solution to many situations. The state of the application can be viewed as an accumulate operation across all the actions starting with the initial state. We do a series of mutations of immutable objects that are declaratively described by actions. We can use smart data structure that can reuse the object by sharing the common structure among individual instances.

4.1.4 Sagas

We also have to explain the Redux saga library that takes inspiration from [8]. In this case sagas are used to declaratively describe side-effects in middleware. They are based on ES6 generator functions. In this way the sagas naturally represent state transitions. This can be used to model asynchronous transactions or sequences of user actions.

4.2 React

React is a library for creating user interfaces [9]. It is based on creating components often defined using JSX with syntax similar to XML. The components are a combination of a template and code that express the logic behind and their data dependencies. Each component has to define at least a render method. In this sense the resulting view is a function of properties and state. Properties are the data the component is fed by the application. State can encapsulate some local logic that does not have to be shared with the application. For example, a check box can store whether it is checked in its state.

4.2.1 Functional Components

React encourages one way data flow and currently supports functional components. They are pure functions of the properties passed as parameters. Their return value is then used as the resulting view. Such components are concise, comprehensive and focus solely on presentation.

4.2.2 Virtual DOM

One of the benefits of using React is that it maintains a virtual DOM structure in javascript. This means that when React application starts the component syntax is parsed and transformed into a tree-like structure in memory. This virtual DOM is then transformed to the regular DOM. After that, any change of the components is done on the virtual DOM. When the modifications are done, the minimal amount of operations to update the regular DOM is computed. The changes are then made based on the computation. This saves performance and makes consecutive DOM operations cheaper.

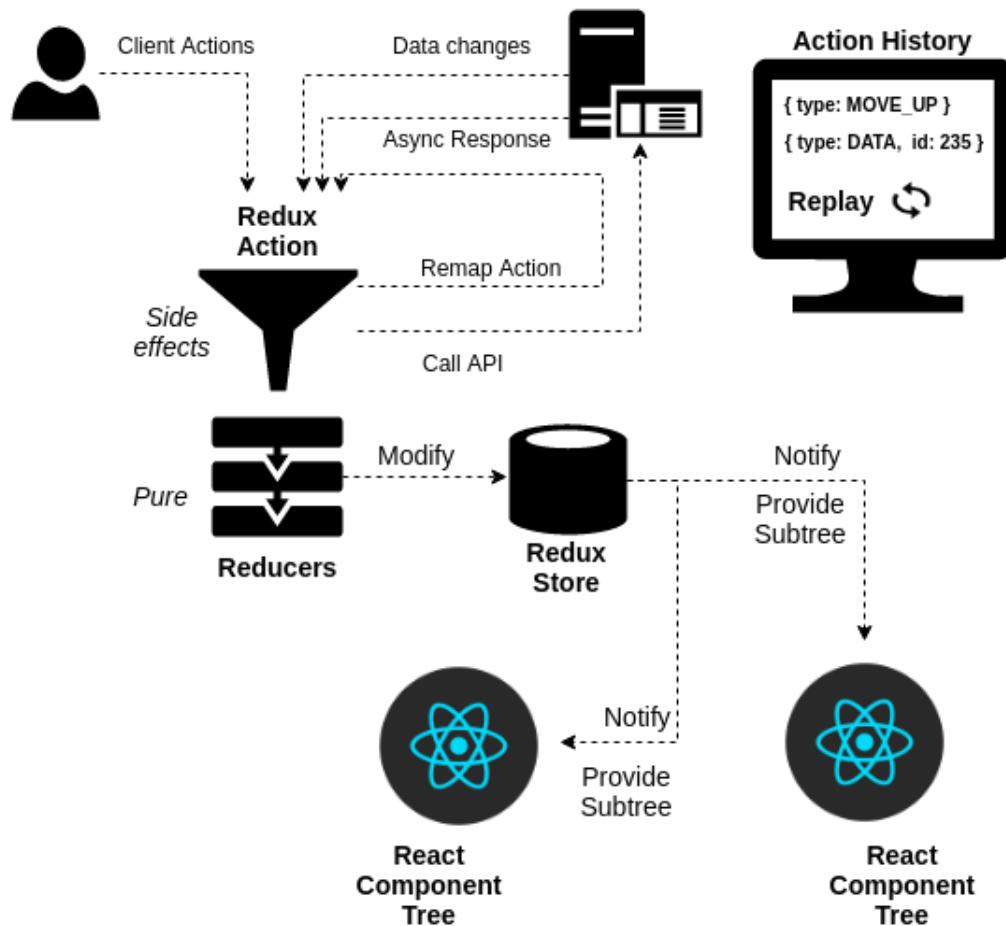


Figure 4.1: React & Redux application structure

4.2.3 Reconciliation

The mysterious computation mentioned above has to discover minimal differences between two trees. The creators of the library were aware that the currently best algorithms have polynomial complexity with respect to tree sizes. This would be too slow for this scenario. Instead they use a heuristic algorithm with linear complexity in number of nodes. They state two assumptions reasonable in practical situations.

- If the nodes have different types, they produce different sub-trees
- Using a unique **key** property, they can determine which two child nodes of the same type should be compared.

If React compares two nodes and they are of a different type it has to destroy the tree and rebuild it. If it they have the same type, only attributes are updated and the instance is reused. After that the algorithm continues with the children. However, we should avoid moving elements across the tree freely as they are are reused only if moved within the same parent.

4.2.4 Optimizations

Another means of improving performance is to prevent any unnecessary computation in the render methods. React components can be supplied by a predicate

which decides if the component is to be recomputed. Generally, pure components that are about to receive the same set of properties need not to be recomputed.

This is not the only case. We can decide that not all of the properties are important enough to trigger a new render. This is the place where we can decide it. On the other hand, this is also useful when we are trying to have a special condition for a render that is not expressed by the properties.

4.3 Webpack

Webpack is a module bundler for javascript applications [10]. It lets us declare our dependencies and do useful transformations. It builds a dependency graph, prepares the modules and then creates bundles to be deployed. To understand Webpack we explain the basic concepts in the order in which they are used at build-time.

4.3.1 Entry

It makes sense that we have to somehow indicate where the process should start. Multiple entry points with different names can be specified. This way we can separate files for more vendors or specify dependencies for multi-page applications. Generally, the entry points are recommended to correspond with individual HTML files.

4.3.2 Output

The next thing that has to be specified is where to put the bundles Webpack prepared. In addition many options related to naming chunks of the bundle or exporting as a library can be used.

4.3.3 Loaders

Loaders are used to transform non-javascript files into modules so they can be included in the dependency graph. They have two essential parts.

- Loaders contain a regular expression under the `test` property. If any file matches the expression it is handled by the loader.
- In order to specify the loader to be used on the files satisfying the test a loader name has to be specified by the `use` property.

4.3.4 Plugins

As loaders can be used only on individual files, plugins are designed to perform actions on the bundle. Before a plugin can be used it has to be required. There is a vast amount of plugins that can solve frequent problems in a simple way.

4.4 EcmaScript 6

The ES6 standard brings many new useful features to javascript. Even though browsers do not currently support the whole standard a transpiling tool, for example Babel, can be used. It takes ES6 adherent code and transforms it to chosen older standard. We do not explain here all the features but restrict ourselves to two of them widely used in the SP project.

4.4.1 Modules

The ES6 module system offers default, named and aliased imports and their export counterparts. Every file can define any number of exported symbols but at most one can be exported as default. This means it can be imported without the need to refer to its name. Any import can be aliased.

```
import React from "react"; // default import
import { newItem } from "../selectAction.js"; // named import
import { update as u } from "../menuAction.js"; // aliased import
```

Listing 4: ES6 imports

4.4.2 Generator functions

Generator functions are the basis of the Saga library. Using a generator we can write readable iterators by defining functions that seem to remember state and return multiple values. They provide a nice syntactic sugar above iterators. Generator functions are marked with an asterisk and can contain `yield` contextual keyword. Any yielded value is immediately returned. When the function is called next time it resumes its execution in that exact place. Listing 5 shows how generator functions in synergy with sagas can be used to capture blocking process in a sequential manner. This function captures a process of creating new network recording which is inherently asynchronous in a declarative style.

```
function* createNewRecording(userId, program) {
  try {
    // return object that requests API call, pause execution
    const record = yield call(Api.newRecord, userId, program)
    // saga resumes execution when the response arrives
    // update Redux store with new recording
    return yield put({ type: 'RECORDING_SUCCESS', record })
  } catch(error) {
    yield put({ type: 'RECORDING_ERROR', error })
  }
}
```

Listing 5: Generator functions usage in sagas

5. Implementation

In this chapter we present the architectural considerations of the SP application. The implementation is described to provide a coherent overview that can serve as a guide to dive into the code. After reading this chapter, any contributor should be able to maintain and extend the application in accord with the original intentions.

5.1 Architecture

The SP application has to be integrated into the platform. It is necessary to fetch data from the server and communicate with portal API Figure 5.1 shows the basic interaction with application server and STB application. It communicates with both the STB portal and application server. The configuration is handled with a separate service. The data layer is separated from the user interface. It is done to separate concerns and simplify development in the long run. A wrapper is needed to adapt the application to be used in the portal. It also has to react to certain life-cycle events. The portal can be run with or without the application making it an enhancement rather than necessary part of the system.

5.1.1 General structure

Here we divide the project into basic logical entities, that can be helpful when reasoning about the implementation.

Configuration service

This service cares about providing consistent information from the configuration files.

Data service

It manages the current state of the data for the application. UI layer is notified about any important change. Additionally it provides access to the external applications and managers.

UI layer

User interface is represented by a large React & Redux SPA that manages the complete UI state in a predictable manner.

Tile modules

They are a collection of files that contain functions serving as various providers for tiles. All of them have to adhere to a simple interface.

External applications

They form a heterogeneous group of applications and managers that integrate existing solutions or encapsulate auxiliary modules.

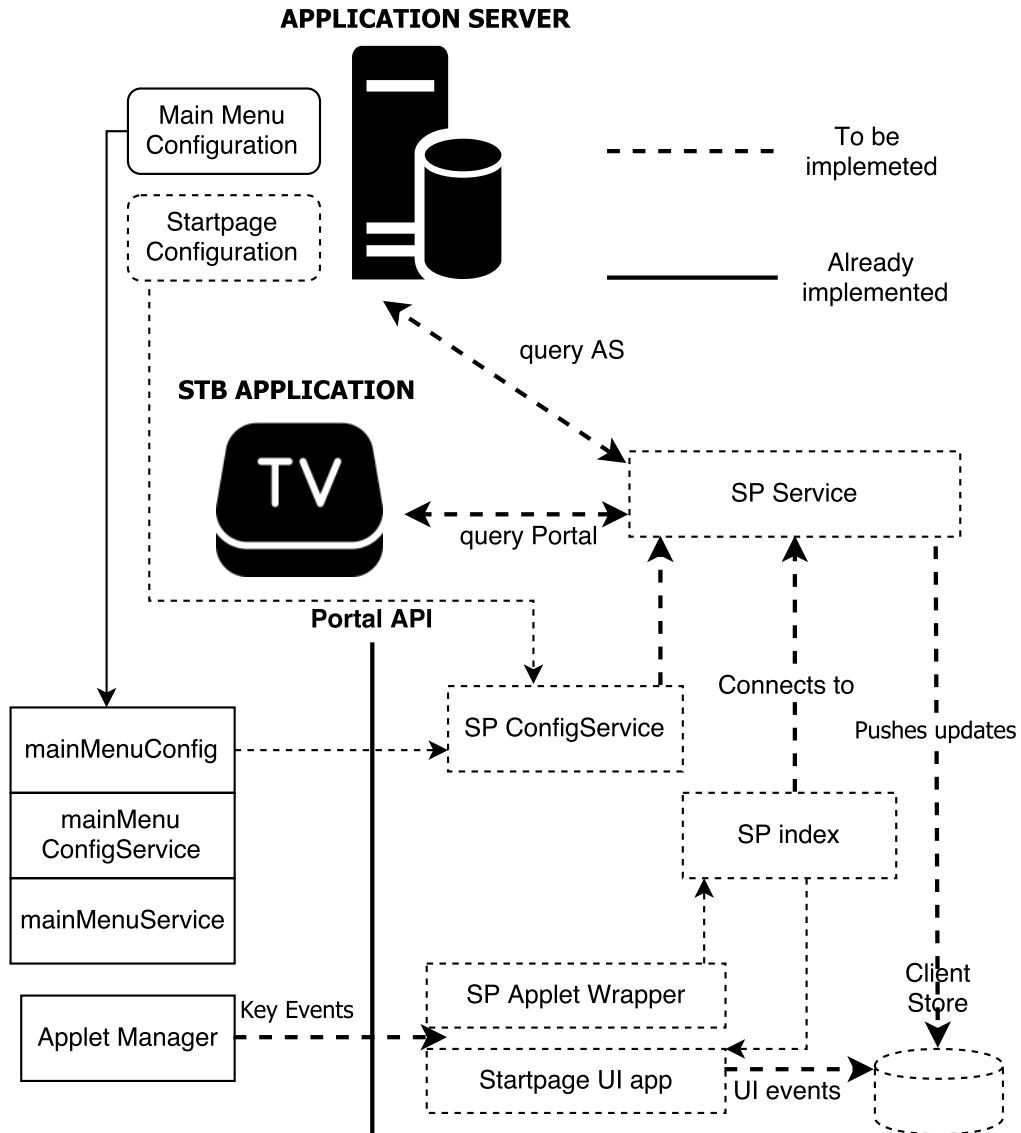


Figure 5.1: Plan for integration with the platform

5.1.2 Files structure

The structure of files follows the dependencies among them. Code is separated into larger number of smaller files. Those which are interdependent are focused together to work as small modules. Emphasis is given to the minimization of distant dependencies. We do not tolerate any circular dependencies.

The system used for the management of modules is the one from the ES6 standard. It is briefly described in the preceding chapter.

In the project root there is a `package.json` file that describes necessary information about the package. It lists external dependencies of the application, some metadata and basic scripts used in development.

Additionally, there is a pair of Webpack scripts used when building the application. The first one is for development uses, with additional tools enabled. The second one is used for production build and contains operations that minify, optimize and bundle the output.

The `src` folder contains folders with parts of the application. Figure 6.4 is included to provide better overview of the file structure.

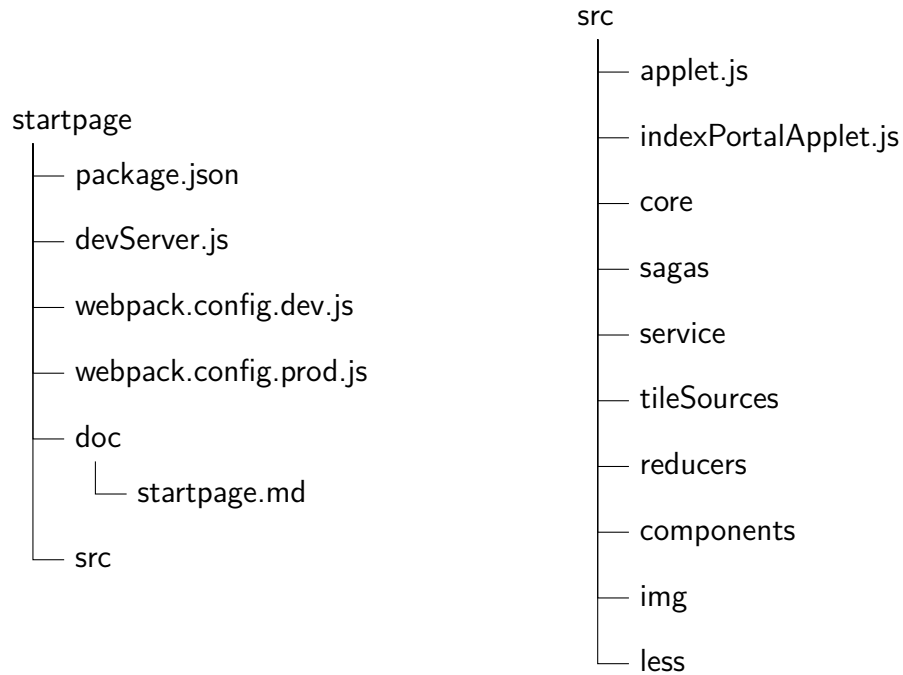


Figure 5.2: Basic directory structure

In the `src` folder aside from source files `img` and `less` folders can be found. The former contains statically defined images used in the application. In the `less` folder all files with application LESS styles can be found.

In the following paragraphs we explain the basic folder structure of our source codes. More detailed description of their contents can be found further in the text.

core

In `core` folder we can find the module that takes care about the creation of our UI layer. The creation of Redux store and composition of reducer functions are also present.

service

The `service` folder contains both the data and configuration services. Along with them we include all auxiliary managers and providers which are used exclusively by the services.

sagas

Here we can find the middleware functions of the Redux store. This includes updates, focus and actions sagas.

tileSources

In `tileSources` folder we put the generic code that handles behaviour of tiles and the registration of a tile source module to the collection of available modules. Two important subfolders are `multiTiles` and `singleTiles`. The former folder

contains modules capable of producing variable amount tiles. The latter consists of modules that produce at most one.

reducers

Our application uses reducer functions to modify its state. All of them are stored in the `reducers` folder. Each important reducer has its own folder. It contains the action creators tied with the reducer, selectors that can query the state managed by the reducer. Naturally, it also contains the reducer itself.

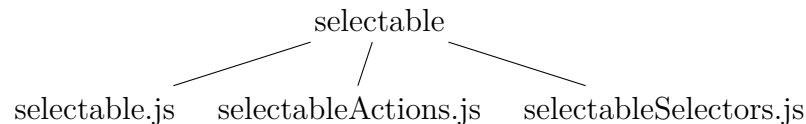


Figure 5.3: Example of a folder in `reducers`

components

In `components` folder we can find our React components. If the classification makes sense they are divided into the subfolders `grid` and `menu`. The rest is directly in the `components` folder.

5.2 Configuration service

The configuration service is a simple module that separates the rest of the application from the concern of having a valid configuration. It offers a simple interface where we can observe any change in the configuration relevant for the SP. This service listens to the service providing the structure of main menu and enhances it with its own applications. It also manages the SP configuration JSON and implements a retry mechanism. In case of an unsuccessful request the application is provided with a consistent offline configuration. Meanwhile the request is repeated in an interval of exponentially increasing length.

```
// returns unsubscribe
addMenuConfigObserver: function(observer) {
  return setObserver(menuObservable, observer);
},
// returns unsubscribe
addGridConfigObserver: function(observer) {
  return setObserver(gridObservable, observer);
}
```

5.3 Data service

We had to separate the logic of the UI from the integration with the rest of the system. The data service provides a core of the application that adapts to the change of configuration. The application can be completely reloaded with a different configuration or transition to an offline state at runtime.

5.3.1 Sources manager

Sources manager is an important module encapsulating the data structure managing the whole grid of tiles. It can also be used to invoke life-cycle events of the tiles, signaling sleep, wake up, refresh or disposal. Freeing the resources used by tiles is crucial as not doing so can lead to memory leaks. The sources manager internally uses a simple cache for tile objects.

5.3.2 Tiles cache

The grid structure provided by the sources manager is only a view on the data structure. Each tile has its own identifier and its functionality is stored only in the cache. However, tile views with smaller footprint can be manipulated freely. They have all the attributes needed for the UI. When an action is invoked on the grid, an identifier of the target tile and type of action are passed to the sources manager. Then the corresponding tile object is found in the cache and the action is invoked from the data service.

5.3.3 Managers and applications

The last important capability of the data service is providing managers for external functionality and abstraction from applications that can be included dynamically into the menu.

Menu items provider

Menu items received from the service are in a different format than required by the SP. This module adjusts them to the right structure and adds menu sections that activate embedded applications.

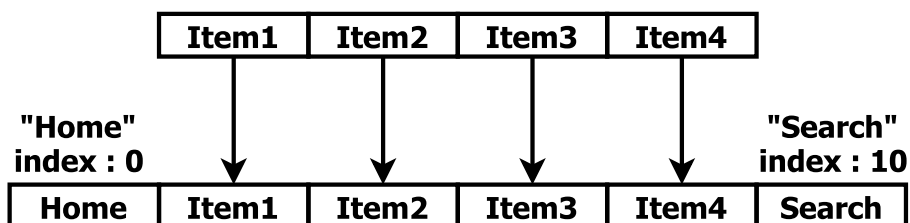


Figure 5.4: Menu enhancement

Banner manager

Banner manager is a simple component that provides a DOM element that can render any banner from a collection. It provides a callback that takes an index as a parameter. When called, it changes the banner rendered in the element according to the specified index.

Search application

The search application is used to find lookup entities across the whole platform. A suitable implementation was available, but it was implemented as a full screen application. We have written an interface, that enables using applications in an embedded form as a part of the SP. They just have to implement the required methods so they can be directed by the menu. We will focus more on the description of menu applications interface later.

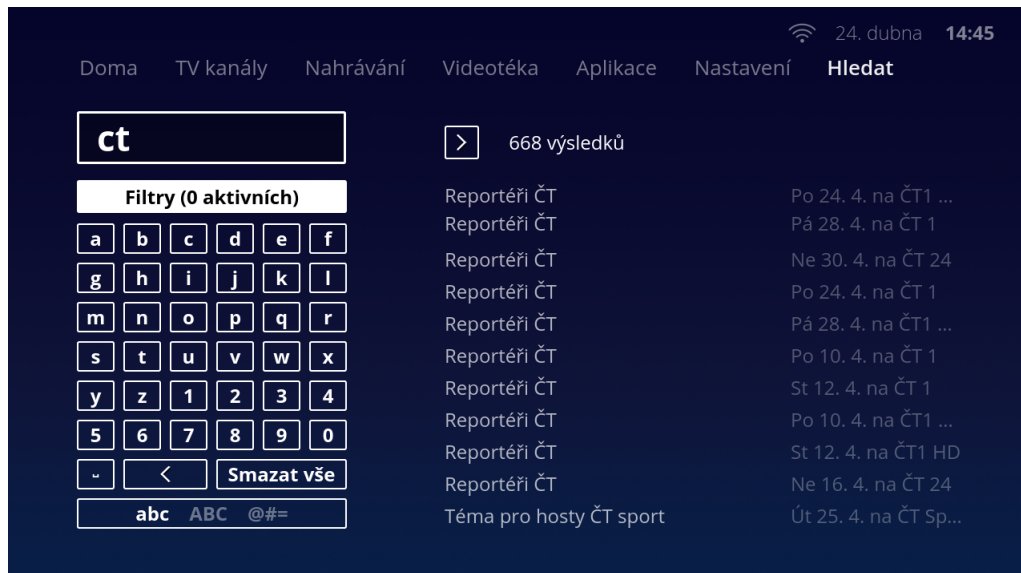


Figure 5.5: Search applications – screenshot

Native application

The tile grid is partly a menu application and should implement the interface. However, it is also managed by the Redux state. From the menu point of view it is part of the application so it is not registered externally.

5.4 User interface layer

The Redux store stores the whole user interface state. It is a simple javascript object that can be modified only by a selected set of actions. It follows the structure of reducers. In our case it stores mainly the tiles in an ID store. Rows, which are arrays of tile references, are also stored by IDs. Grid is just an array of rows to be displayed. Besides that, the store contains information about current focus and it stores the structure and state of the textual menu.

5.4.1 Reducers

Reducers can be written in a specialized manner or to be more universal. We decided to write universal reducers and leverage function composition to bend them to fit the specific case.

Root reducer

In Figure 5.6 we can see how the reducers are composed into one root reducer which handles all the state modifications.

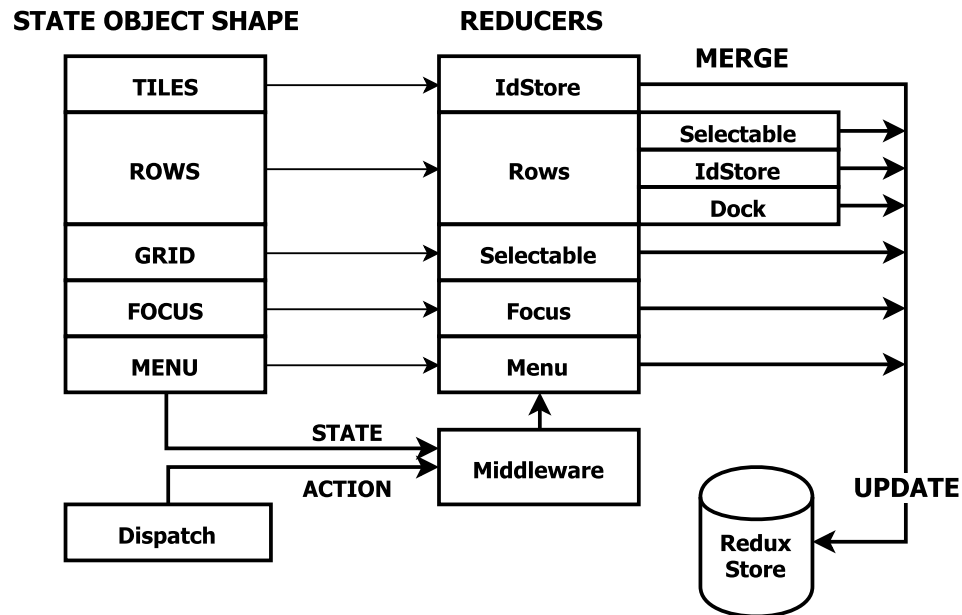


Figure 5.6: Reducers structure and workflow

The root reducer is a composition of our reducers. Generally, for each top level key in the Redux store we have a single reducer managing the sub-state. There is no restriction on composing the reducers as long as they remain pure. We also enhance the root reducer with two mechanisms.

The first enhancement performs batching of multiple actions. If an action of `BATCHED_ACTIONS` type is dispatched, it is not passed to the reducers directly but actions from its `actions` property are dispatched instead. This means that all state changes in this batch are passed to listeners as one.

The second one enables domain specific actions. If an action has a property `domain` defined it is compared with the top-level keys in the store. The action is then handled exclusively by the reducer attached to that key.

Concepts

We can, for example, notice that the `selectable` reducer is used twice. For the first time, we use it as the reducer managing the grid state. In the second case the `selectable` reducer is used as a part of the reducer handling rows.

In this manner we capture the similarity in the navigation within rows and between them. We create a logical concept of `selectable` that can manage a

state of a defined structure and execute actions to modify it. In Figure 6 we provide an illustration of the `selectable` concept.

```
/* Available actions:  
    SELECT(index)  
    NEXT  
    PREVIOUS  
    INSERT_ITEM(index, item)  
    REMOVE_ITEM(index)  
    NEW_ITEMS(items) */  
initialState = Immutable({  
    items: [],  
    selected: 0  
});
```

Listing 6: Selectable illustrated

While in grid the `selectable` concept is applied directly on the management of the collection of rows, the rows themselves use a more complicated mechanism. The row reducer also cares about storing the individual row data under unique identifiers using the `idStore` concept. Besides that, we also use a small `dockReducer` to maintain the right shift of the row. It depends on the item selected, the geometry of the tiles and the process by which we got to the state. Nevertheless, the algorithm is quite simple. After each state change we check whether the newly selected item is completely in the specified visible area. If not, we move shift the whole row in the direction that moves the tile to the visible area. The length of the shift is computed as the minimal shift that gets the entire selected tile to the visible area.

As we mentioned, the `idStore` reducer works as a traceable ID map. The menu reducer handles a two dimensional array that can be refreshed and traversed in a circular way. The focus reducer stores simple information about current focus. Focus is handled mostly in middleware functions. The reasons why we do not use reducers for this functionality is described in the next section.

5.4.2 Middleware

Middleware functions are functions that are called on actions before reducers, but they do not modify the state. They can solve some problems that are difficult to handle in reducers. We continue with a list of notable features of middleware in comparison with reducers.

- In middleware we manage with side effects. They are the place where we can interact with external code or call server APIs.
- In middleware we can dispatch actions. This is a nice way to handle asynchronous operations. After receiving some user initiated we call the API, but do not handle the original action in reducers. When the request is received we create a new action telling the reducer to adjust the state to the new data.

- Middleware can query the whole state while reducers work only on its subset. This gives them complete information about the application.
- Middleware is a great extension point. Many libraries providing middleware functionality exist.

In our middleware we use the Redux Sagas library. It is based on composition of few generator functions. Each of them tries to pattern match the action. In case of success it can handle the action with read-only access to the state. We create three instances of saga middleware to handle data updates, actions with external impact and focus.

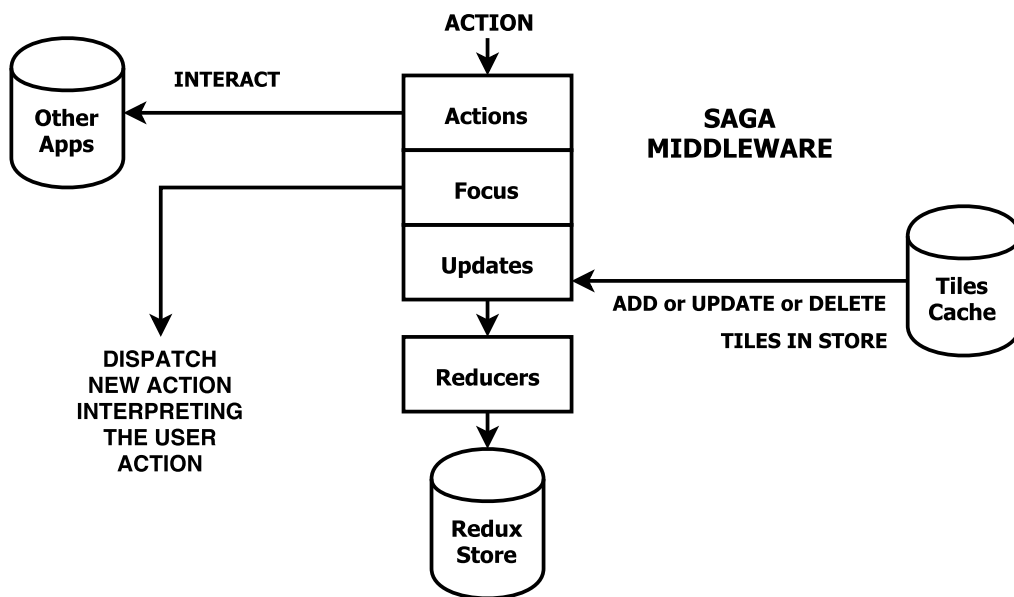


Figure 5.7: Middleware usage

Focus

The focus saga exists because of the non-contextual nature of RC input. Unlike traditional mouse inputs and touch inputs, our input does not include information about position. That means we have to track the state of the focus. In our application there is a large amount of focusable entities on the screen. We decided to track separately which top-level component is currently focused. This enables us to decide which component should be notified about the action.

It is important to realize how to direct focusable components. We designed them so that they understand certain events specific to them. This means we have to take care of several issues.

- We have to translate events without context to events specific to top-level components. For this translation we need to know the target. To state an example, `KEY_LEFT`, when the focus state is `menu`, is translated to `PREVIOUS` action within the `menu` domain.

- The transfer of focus between two top-level components cannot be solved internally. Every such component has to either provide a method for deducing whether the component will lose focus on a certain key event or provide its state a let the director decide. The focus middleware uses the latter while menu applications have to provide a `canLoseFocusOn` method.
- We query application state to decide how to interpret the action. For example when moving between rows we have to know details of the row losing focus to focus the right tile on the row that is being focused.

```
function* moveBetweenRows(action /* KEY_UP or KEY_DOWN */)
{
  center = yield select(selectCenter);
  yield put(enhanceAction(action, { domain: "grid" }));

  rowId = yield select(selectedRow);
  tilePosition = yield select(selectTileAtPos, center);

  yield put(enhanceAction(
    Selectable.actions.select(tilePosition),
    { domain: "rows", id: rowId }
  ));
}
```

Listing 7: Focus saga - movement between rows

When an action is fired the focus saga middleware interprets the action in context of the currently focused component. In Listing 7 we can see how `KEY_UP` and `KEY_DOWN` arrow keys are handled when the grid is focused. First, we query the state for the center of the currently selected tile. Then we dispatch action for the grid to change selected row. Ultimately, we find the tile below the formerly focused tile and instruct the newly focused row to focus it.

Updates

Updates saga takes care about translating updates from the data service to Redux actions. For the updates saga we decided to include simplified excerpt from the source code. In Listing 8 we present the events handled by the updates saga.

Actions

The actions saga handles interaction with other applications. When an action of channel tile is invoked the channel should be opened in player. Movie tile should lead to its detail. This is done through actions saga. Key events are examined and if an action for the selected tile is found for that key, it is executed. One tile can handle multiple actions.

```

switch (e.event) {
  case ROWS_CACHE_CHANGED:
    yield put(enhanceAction(
      performUpdates(transformToRows(e.toAdd), e.toDelete),
      { domain: "rows" }
    ));
  case TILES_CACHE_CHANGED:
    yield put(enhanceAction(
      performUpdates(e.toAdd, e.toDelete),
      { domain: "tiles" }
    ));
  case ROW_REFS_CHANGED:
    yield put(enhanceAction(
      newItem(e.references),
      { domain: "rows", id: e.targetId }
    ));
  case ROW_REF_CHANGED:
    yield put(enhanceAction(
      replaceItem(e.oldId, e.newId),
      { domain: "rows", id: e.targetId }
    ));
  case GRID_REFS_CHANGED:
    yield put(enhanceAction(
      newItem(e.items),
      { domain: "grid" }
    ));
  case MENU_CHANGED:
    yield put(menuUpdate(e.items));
}

```

Listing 8: Simplified excerpt from updates saga

5.5 Menu applications

We have built in a mechanism into the focus saga which manages switching between different menu applications. We illustrate how it works on the example of the search application.

Search is an application built in a framework that uses the MVVM architecture. We decided that the best way to utilize it is by appending the compiled view to a React component. That way we can manage the view and easily modify it.

To direct it we take the model that paired with the view and write an adapter to our interface. It has to be able to accept signals and translate our actions to events understandable by the model.

Handling signals

Menu applications have to react correctly to several signals. These are `focus()`, `unfocus()` and `init()`. In the case of search, on `focus()` we have to select the right button. Handling `unfocus()` is more complicated.

Search can open other applications within it. There are two another levels of hierarchy that can be opened when using search filter. We have to take care of closing them properly when leaving the search application. Lastly, `init()` cares about getting the model to a consistent initial state.

Translating events

When the application is focused we have to provide it by key events. It cannot be done directly for two reasons. We have to translate the events to the right form and check whether we want to take focus back. The translation in our case is straight-forward, we just take pass the original key events. The function name is `passKeyEvent(event)`.

Losing focus

To know when to leave the application we implement `canLoseFocusOn()` function. It should return the set of events that lead to losing focus under the current state of the model.

The manager provides two additional getters `getEl()` and `getClasses()`. They are used to get the view and additional classes to use for this instance respectively.

5.6 Tile sources interface

The content of our application is presented mainly via the homogeneous tiles. Our implementation has been chosen to have the following properties. All of those functions are non-parametric.

- Every tile source is an independent module. It has to be registered as an available module and can it be used by creating a new configuration object in the JSON configuration.
- The modules communicate via universal interface. It specifies the contract between individual tile sources and the service.
- The modules are responsible for freeing all resources they hold. The service, on the other hand, is responsible for notifying them when they cease to be managed by the service.

5.6.1 Contract

The individual modules should export a single function that returns API of the tile source. The object returned has four optionally defined functions.

Unsubscribe

The tile source is required to get rid of all references to it that exist in the application. This means mainly that the modules have to take care about disposal of any observers they registered. Due to the nature of javascript runtime, no object for which a reachable reference exists can be garbage collected.

Refresh

The tile sources can define an explicit way to refresh information from data sources they use. For that case a refresh method is to be defined.

Sleep

This method is called when SP is ceasing to be present on the screen. Tile sources can stop unnecessary operations when inactive.

Wake

Tile sources get a wake notification when they should recover from inactivity. It should perform operations inverse to sleep.

There are another two, even more important, parameters of the tile source constructor function. The first parameter `sourceConf` is the configuration taken from the JSON configuration file. It contains the essential information for the creation of tiles. The second parameter `acceptTilesCallback` is a function provided by the service.

The service maps each callback it provides to a certain place in the grid structure. The configuration grid consists of sources structured in rows. The visual grid is constructed by the tiles provided by the sources at the places defined by the mapping.

Name	Type	Notes
<code>description</code>	string	Text under tile.
<code>bigLogoSrc</code>	URL	Large overlay logo.
<code>logoSrc</code>	URL	Small overlay logo.
<code>progress</code>	number (0,1)	Shows progress bar.
<code>progressCaption</code>	string	Shows progress bar caption.
<code>whiteBadge</code>	string or number	Red badge with value.
<code>redBadge</code>	string or number	White badge with value.
<code>overTitle</code>	string	Shows title as overlay
<code>overTime</code>	string	Other overlay. Originally time.
<code>overLogoSrc</code>	URL	Shows logo as overlay.
<code>recording</code>	boolean	Show recording indication.

Table 5.1: Tile sources option parameters

5.6.2 Producing tiles

To push tiles to the service through `acceptTilesCallback` function has to be called with the produced tiles as parameters. In Figure 9 we list the possible

overloads of the callback. The first option is to send an individual tile as output and it replaces current tiles. The second option is used to replace tile on a specified index. The caller is responsible not to use it on a non-existent index within the source. The last one is used to replace all tiles of that source with another set of them.

```
acceptTilesCallback({ TILE }) // single tile  
acceptTilesCallback({ TILE }, indexToReplace) // replace  
acceptTilesCallback([ { T1 }, { T2 }, ... ]) // multiple tiles
```

Listing 9: Overloads of tiles callback

The tile objects pushed to the service can make use of two properties of object type. Firstly, **actions** can contain functions as properties. Their key serves as an identifier of the type of key event that invokes the action. Secondly, **options** object can contain various properties from a predefined set.

In Table 5.1 the possible properties of the options object are listed. All of them are implemented visually and are ready for usage by any tile source.

6. Evaluation

Our application targets currently used STB devices. We tested our implementation both on STBs and PCs. On STBs we mainly checked the results while on PC we did a more detailed analysis trying to detect and resolve specific problems.

To evaluate the performance we used basic time measurements and Chrome profiling tools. They proved useful for checking many quantitative and qualitative aspects of our implementation.

6.1 Browser Profiling

A useful tool for checking if the actual renders are minimal is the rendering tab in Chrome developer tools. Paint flashing easily uncovers which elements are rendered when an action occurs. It also provides frame rate measuring, distribution graphs and processor usage.

Layers tab can provide a high-level overview of the DOM structure. It helps with finding redundant elements and computing aggregate paint operations. Another feature is a three-dimensional view on the DOM. That helps with analysis of stacking contexts and z-index property.

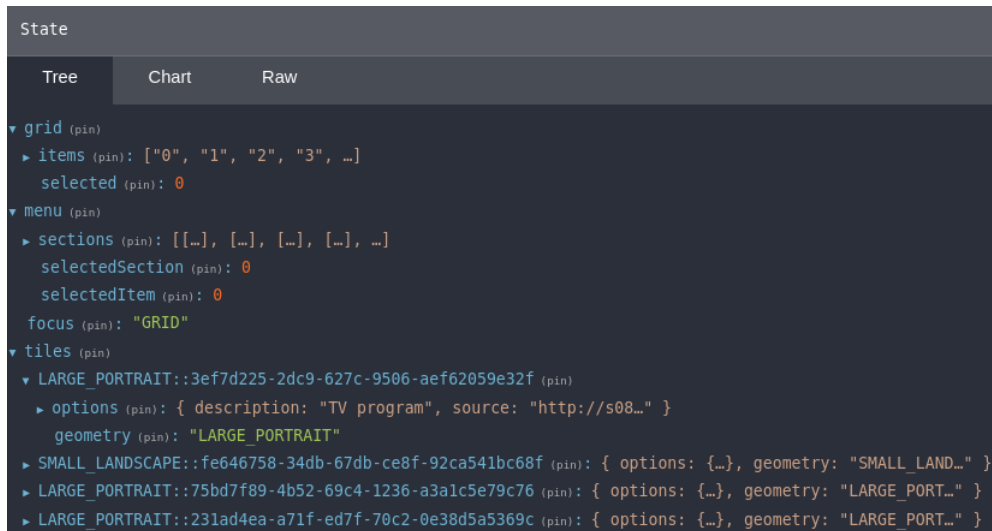


Figure 6.1: Layers tool at work

Profiling tab enables taking heap snapshots, control memory allocation and perform classic profiling. We used it to check for memory leaks by comparing heap snapshots over larger time intervals.

6.2 React & Redux developer tools

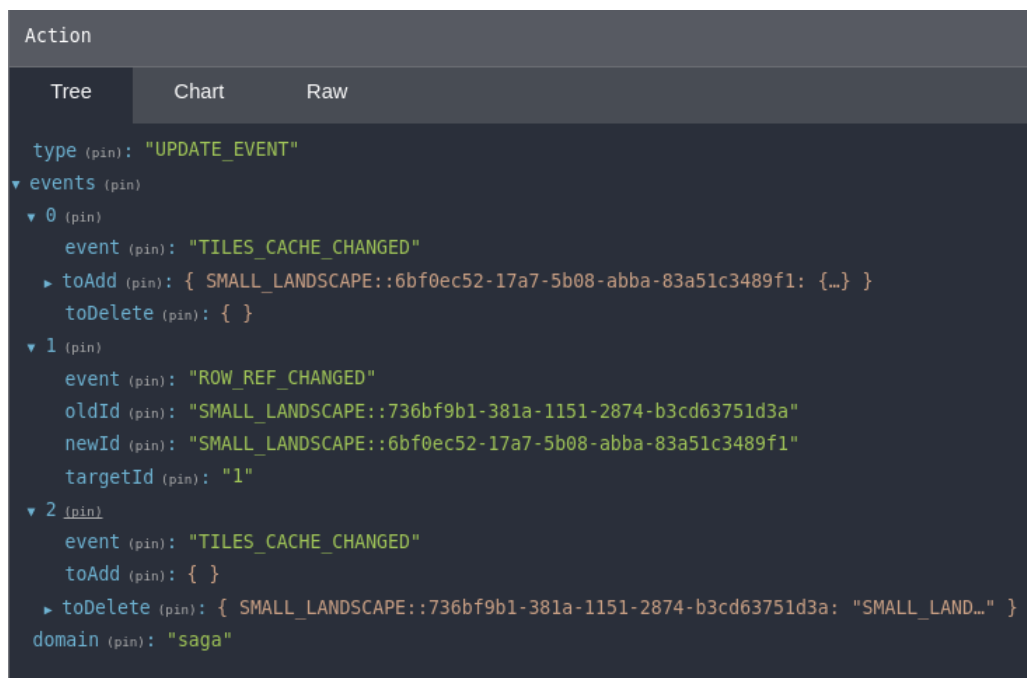
During implementation, we used specialized developer tools for React & Redux. The Redux tools were invaluable as they enabled us to track all partial state changes of our application. We can inspect how the action looks like, the prior and posterior state, and of course the changes made by the reducers. Detecting bugs can be much easier, as we just check whether the action resulted in the expected state mutations.



```
State
Tree Chart Raw
grid (pin)
  items (pin): ["0", "1", "2", "3", ...]
  selected (pin): 0
menu (pin)
  sections (pin): [[...], [...], [...], [...], ...]
  selectedSection (pin): 0
  selectedItem (pin): 0
  focus (pin): "GRID"
tiles (pin)
  LARGE_PORTRAIT::3ef7d225-2dc9-627c-9506-aef62059e32f (pin)
    options (pin): { description: "TV program", source: "http://s08..." }
    geometry (pin): "LARGE_PORTRAIT"
  SMALL_LANDSCAPE::fe646758-34db-67db-ce8f-92ca541bc68f (pin): { options: {...}, geometry: "SMALL_LAND..." }
  LARGE_PORTRAIT::75bd7f89-4b52-69c4-1236-a3alc5e79c76 (pin): { options: {...}, geometry: "LARGE_PORT..." }
  LARGE_PORTRAIT::231ad4ea-a71f-ed7f-70c2-0e38d5a5369c (pin): { options: {...}, geometry: "LARGE_PORT..." }
```

Figure 6.2: Application state in Redux developer tools

In Figure 6.2 we can see the structure of our application state. Grid is a simple list of IDs of currently displayed rows. Similarly, each row references individual tiles managed by the `tiles` reducer, which uses the `idStore` concept.



```
Action
Tree Chart Raw
type (pin): "UPDATE_EVENT"
events (pin)
  0 (pin)
    event (pin): "TILES_CACHE_CHANGED"
    toAdd (pin): { SMALL_LANDSCAPE::6bf0ec52-17a7-5b08-abba-83a51c3489f1: {...} }
    toDelete (pin): { }
  1 (pin)
    event (pin): "ROW_REF_CHANGED"
    oldId (pin): "SMALL_LANDSCAPE::736bf9b1-381a-1151-2874-b3cd63751d3a"
    newId (pin): "SMALL_LANDSCAPE::6bf0ec52-17a7-5b08-abba-83a51c3489f1"
    targetId (pin): "1"
  2 (pin)
    event (pin): "TILES_CACHE_CHANGED"
    toAdd (pin): { }
    toDelete (pin): { SMALL_LANDSCAPE::736bf9b1-381a-1151-2874-b3cd63751d3a: "SMALL_LAND..." }
domain (pin): "saga"
```

Figure 6.3: Saga events in Redux developer tools

The tools can be used to debug complicated actions. Figure 6.3 shows an `UPDATE_EVENT` action received from the data service. It instructs the store to remove old tile from the `idStore` and replace it with a new one also refreshing the references. This update was a result of a new programme starting on one of the channels from `nowInTv` tile source.



(a) Main component – properties (b) Row component – properties

Figure 6.4: React developer tools

React developer tools can be used to track the changes of the virtual DOM. They also provide similar debugging tools for the virtual DOM as Chrome does for the regular one. Figure 6.4 contains properties list of two components. The main component integrates the external functionality and contains the grid and menu components. Row component receives all the state necessary to render it from the Redux store.

6.3 Rendering Optimality

During prototyping we found that a naive approach that recomputed the whole virtual DOM each time the state changed is not viable. One action took more than half a second when there were many tiles on the screen. We had to restrict the computation only to components that had to be rendered again.

We used Seamless immutable library that provides immutable structures with API of plain javascript objects and arrays. We only had to replace classical mutations with immutable merges. In the component tree we added check for reference equality that detected if the properties changed from the last time and thus have to be rendered. Thus components which receive the same reference on a part of the immutable structure are not recomputed. The immutable structures from store persist the transfer through selectors and are passed to the individual components from the root component.

6.4 Measuring the effect

We have done some benchmarks on the main targeted STB model featured in the first chapter. To analyse the effect of render optimizations that became viable with immutable state, we decided to measure the duration of individual renders.

We identified the number of tiles as the key factor for performance. We do not expect the page should contain more than a hundred tiles. On the other hand, we recommend, for visual reasons, that there are at least thirty tiles in at least four rows.

Immutable structure	Mean	Standard Deviation
No	256.8ms	40
Yes	118.3ms	23.6

Table 6.1: Action event duration – 30 tiles, 100 observations

We built a production version of the application with four different setups. At first, we removed the immutable optimizations and provided a configuration with thirty and then with one hundred tiles. We designed our benchmark so that it is similar to our use cases. In total we measured 100 actions in each setup. Exactly 10 of them were a movement between rows. The remaining ninety moves were made within rows rotating through the carousel.

We repeated the same process with the optimizations. In Figure 6.1 we can find results for the smaller setup. We see a significant improvement with the immutable checks. The comparison with the larger setup, however, is more interesting. While in the case without immutable we can see linear relationship between the render time and amount of tiles, this is not true for the optimized case. In Figure 6.2 we see that the unoptimized setup took more than thrice the time in average. On the contrary, the optimized one took only 30% more time. That is approximately ten times lower slowdown.

The vertical movement showed to be approximately two times slower than the movement within row. That is because all rows on the page change position when moving between rows.

Immutable structure	Mean	Standard Deviation
No	807.4ms	132
Yes	155.2ms	44

Table 6.2: Action event duration – 100 tiles, 100 observations

6.5 Player in picture

Two of the STB models that we are required to support have issues with using player in picture functionality. The video plays in a special element behind the web browser. Using the API function that in fact cuts a rectangular hole through the DOM made the STB slower by a degree. So we were forced to find another solution.

As we were unable to get the player to foreground we had to ensure that there will not be any non-transparent element in front of it. This had to be incorporated into the DOM structure. Essentially, it meant that we had to focus on two problems.

- In any moment there cannot be any tile or other visual object covering the player. This has shown to be easy to satisfy, as the player is a tile in itself and there was no need to cover it. Only thing to be done was to make the tile produced by the player tile source transparent.
- We have to maintain a transparent rectangular space in the background gradient. That is challenging, as the player does not have a fixed position. It has to be moved as the user navigates himself within the grid.

We arrived at the conclusion that we have to split the background into four parts which will stretch around the player and together form the background gradient. Background parts not only change dimensions as the player moves but they also have the background offset recomputed to maintain consistency of the gradient. This approach showed to work significantly better than the API function.

7. Conclusion

As an integral part of this thesis we implemented a complete startpage application for TVs that runs on set-top box devices. We have successfully accomplished the required functionality, both from the administrative and functional point of view. All tile source modules are ready to use, and have some additional configuration options beyond the requirements.

7.1 Implementation assessment

The interface we proposed for the tile source modules showed as simple and flexible. We extended it to enable modules to optionally implement life-cycle callbacks. We also implemented an universal mechanism to embed custom applications into menu sections.

We consider the usage of React and Redux libraries as highly beneficial to our effort. The code is well structured and highly declarative. The whole application state is stored in an immutable structure and managed only by pure functions. We can say that the views are just a pure function of the state and is inherently stateless.

We implemented rendering optimizations, leveraging the immutable state. Without them, the application would be too slow to be used on the targeted devices. We used Redux Saga library to handle non-contextual actions in a concise way. Webpack provided seamless build, hot-reloading and bundling. We extensively used new features of ES6, mainly generator functions and modules.

Startpage can be parametrized by the configuration JSON that clearly defines the structure of the grid and menu applications to be used. The configuration can be changed at runtime, disposing the old resources. The grid is then restructured according to the new configuration. Overall, we consider the goals fulfilled.

7.2 Future work

The application is considered as completed. However, two possible extensions come to mind.

Create new tile sources

New tile sources can be used to make more functionality accessible from the Startpage. Whether it be a shortcut to settings or recommended movies listing, the interface will make it simple to write a new module.

Integrate new menu application

New applications can be integrated into the menu, for example weather or news overview.

At some point it may not be sufficient to configure the Startpage structure globally. Individual users could add their own tiles to the grid from a predefined set. Other applications of their choice could be made accessible through the menu. Eventually, they would be able to individually manage they TV home screen as on their mobile devices.

Bibliography

- [1] Tom Butts. The State of Television, Worldwide. <http://www.tvtechnology.com/opinions/0087/the-state-of-television-worldwide/222681>, December 2013. Accessed: 2017-05-05.
- [2] Ali Mesbah and Arie Van Deursen. Migrating multi-page web applications to single-page ajax interfaces. In *CSMR '07 Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, pages 181–190. IEEE, 2007.
- [3] Emmitt Scott. *SPA Design and Architecture: Understanding Single Page Web Applications*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2015. ISBN 1617292435, 9781617292439.
- [4] Michael S. Mikowski and Josh C. Powell. *Single page web applications*. Manning Publications, 2013.
- [5] Eugeniya Korotyia. 5 best javascript frameworks in 2017. <http://da-14.com/blog/5-best-javascript-frameworks-2017>, January 2017. Accessed: 2017-03-23.
- [6] Eric Elliott. Top JavaScript Frameworks & Topics to Learn in 2017. <http://medium.com/javascript-scene/top-javascript-frameworks-topics-to-learn-in-2017-700a397b711>, December 2016. Accessed: 2017-03-23.
- [7] Redux - Introduction. <http://redux.js.org/docs/introduction/>, January . Accessed: 2017-04-26.
- [8] Hector Garcia-Molina and Kenneth Salem. *Sagas*, volume 16. ACM, 1987.
- [9] React. <http://facebook.github.io/react/>. Accessed: 2017-04-26.
- [10] Webpack. <https://webpack.js.org/>. Accessed: 2017-04-27.

A. Documentation

This appendix serves as a guide for the administrator who is in charge of configuring the application and its content. It also lists API dependencies of the individual tile sources.

A.1 Basic configuration

Server configuration

The `sws` configuration has to contain the keys in table A.1.

Key	Type	Explanation
<code>startpage.enabled</code>	boolean	Enable or disable startpage
<code>startpage.config.jsonUrl</code>	URL	Startpage configuration
<code>mainMenu.config.jsonUrl</code>	URL	Menu configuration
<code>banners.url.startpage</code>	URL	Banners API URL

Table A.1: Server configuration – required keys

Menu section banners

Menu sections in startpage can have different interactive banners. In case of `banners.url.startpage` containing URL of an image all sections will share the same banner. In case of using some API the request for will contain parameter `applet=startpage:[SECTIONID]` where section `[SECTIONID]` is id of the corresponding category in `mainMenuConfig`.

Configuration JSON

The configuration file must contain a JSON with the following structure:

```
{
  "version": "1.0",
  "addToMenu": <menu-definition-json>,
  "grid": <tiles-definition-json>
}
```

Listing 10: Configuration top level structure

Additional menu sections

Startpage displays the main menu, taken from `mainMenuConfig`, in its top row. This can be extended with few startpage specific applications under added menu sections. Currently supported are:

home Displays the graphical tiles.

search Search applet embedded into Startpage. It follows the visual style and controls of the new user interface.

```
"addToMenu": [  
  {  
    "insertAt": 0,  
    "appId": "home",  
    "titles": {  
      "eng": "Home",  
      "cze": "Doma"  
    }  
  },  
  {  
    "insertAt": 10,  
    "appId": "search",  
    "titles": {  
      "eng": "Search",  
      "cze": "Hledat"  
    }  
  }  
]
```

Listing 11: Menu enhancement definition

A.2 Grid configuration

The tile grid is specified as a list of rows. Each element of the array defines one row. Row has array of sources where each element is an tile source definition object.

A.2.1 Row definition

Each row has the following parameters:

All `TILE DEFINITION` objects have similar form and common parameters listed in Table A.3.

```

"grid": [
  {
    "height": "large",
    "titles": {
      "eng": "Applications",
      "cze": "Aplikace"
    },
    "sources": [
      TILE DEFINITION1,
      TILE DEFINITION2,
      ...
    ]
  },
  ...
]

```

Listing 12: Row definition

Name	Type	Notes
height	string	Either <i>large</i> or <i>small</i> .
titles	JSON	Optional. Titles in JSON.
titles	array	List of <i>tile definition</i> objects described below.

Table A.2: Row parameters

A.3 Available tile types

The tiles usually carry additional parameters, which are specified in the following description of all possible tile types.

Frequent portal API dependencies

```

sws
    .call
    .getAbsoluteMediaUri
Logger
format

```

A.3.1 Types producing multiple tiles

This section lists all the tile sources that are able to produce multiple visual tile objects.

Now on TV

Displays multiple tiles in a row. Each tile corresponds to a TV channel in the currently active channel list. It displays the TV programme currently running on

```

"type": "someTileType",
"orientation": "portrait",
"tiles": {
  "eng": "English Title",
  "cze": "Český titulek",
  ...
},
...

```

Listing 13: Tile source objects

Name	Type	Notes
type	string	Required. Tile types are in the following sections.
orientation	string	Tile orientation, either “landscape” or “portrait”.
titles	JSON	Optional. Localized variants.

Table A.3: Common tile parameters

that channel. The tiles link to the TV player, switching playback to the respective channel in "live" mode. Optionally, the tiles can open TV program detail page of the program they display.

```

"type": "nowOnTV",
"maxTiles": 12,
"play": true

```

Listing 14: Now on TV – Example

Actions

keyOk Play the currently selected programme and channel.

keyRed Record the currently selected programme.

Portal API dependencies

```

channelService
  .getChannelListChannels
  .getChannel
tvApp
  .getCurrentChannelListKey
  .addChannelListChangeObserver
  .removeChannelListChangeObserver
  .play
pvrService

```



```

        .addRecordingObserver
        .removeRecordingObserver
epg.bufferedEpgService
        .getPFPrograms
pvr
        .RecordUtil
dialogs.utils.entityDetails
        .epgDetail

```

Name	Type	Notes
maxTiles	number	Upper bound on tiles produced.
play	boolean	Go to player/go to detail.

Table A.4: Now on TV parameters

Recordings

Displays multiple tiles in a row, that correspond to last recorded TV programs. The tiles link to the record detail page.

```

    "type": "recordings",
    "maxTiles": 12

```

Listing 15: Recordings – Example

Actions

keyOk Open detail of the recorded entity.

REST API dependencies

/subscription/search/search

Portal API dependencies

```

pvrService
    .addRecordingObserver
    .removeRecordingObserver
dialogs.utils.entityDetails
    .npvrDetail

```

Name	Type	Notes
maxTiles	number	Upper bound on tiles produced.

Table A.5: Recordings parameters

VOD movie category

Displays multiple tiles in a row, which correspond to individual VOD movies that fall into movie category specified by search filter, see parameter description below. Tiles display movie cover images.

Actions

keyOk Open detail dialog of the selected entity.

```

"type": "vodCategory",
"maxTiles": 12,
"params": {
  "genre": [5, 23, 64],
  "providerCodes": ["code"],
  "sortField": "rating",
  "sortOrder": "DESC"
}

```

Listing 16: VOD movie category – Example

REST API dependencies

/subscription/search/search

Portal API dependencies

```

.dialogs.utils.entityDetails
.vodDetail

```

TV, VOD, OPVR categories

This flexible tile type can display one or more tiles in a row, which correspond to individual categories of appropriate *epgCategories*, *vod* or *opvr*, applications which in turn must be configured in the set-top box main menu.

The example shows configuration of three category tiles that link to respective categories of the application *epgCategories*, which must be configured in the main menu. In other words, the main menu configuration must include a menu item.

Name	Type	Notes
maxTiles	number	Upper bound on tiles produced.
params	JSON	Parameters for VOD application
params.providerCodes	string[]	Enumerates enabled providers.
params.sortField	string	Field used for sorting.
params.sortOrder	string	<i>ASC</i> or <i>DESC</i> .
params.genre	number[]	Enumerates enabled genres.

Table A.6: VOD movie category parameters

Actions

keyOk Run the applet specified by the configuration with defined parameters.

Portal API dependencies

```
mainMenu.mainMenuService
    .getItem
portal.mainMenuConfig
    .items
```

Name	Type	Notes
mainMenuItemId	string	Reference to main menu config item
categories	JSON[]	Specifies individual categories to list.
categoryId	string	Enumerates enabled providers.
tileImage	URL	Absolute or relative URL
titles	JSON	Optional.

Table A.7: Categories parameters

A.3.2 Types producing a single tile

Player in picture

This tile source can be used only once per application. It creates a tile with player in picture.

Actions

keyOk Go to player with the current playing entity.

```

"type": "categories",
"orientation": "portrait",
"menuItemId": "epgCategories",
"categories": [
{
    "categoryId": "top",
    "tileImage": "startpage-img/tv/top.jpg"
},
{
    "categoryId": "series",
    "tileImage": "startpage-img/tv/series.jpg",
    "titles": {
        "eng": "Series",
        "cze": "Seriály"
    }
},
{
    "genre": "6",
    "tileImage": "startpage-img/tv/comedy.jpg",
    "titles": {
        "eng": "Comedy",
        "cze": "Komedie"
    }
}
}

```

Listing 17: Categories – Example

Portal API dependencies

```

player.playerApplet.controller
    .switchToPlayerInfo
channelService
    .getChannel
playerApp
    .unsubscribe
    .subscribe

```

Link to EPG

Single tile that links to EPG page with custom title and image.

Actions

keyOk Go to EPG applet.

Portal API dependencies

```

epg.init.run

```

```

"type": "player",
"titles": {
  "eng": "Currently playing",
  "cze": "Právě hraje"
}

```

Listing 18: Player in picture – Example

```

"type": "epgLink",
"titles": {
  "eng": "Program guide",
  "cze": "TV program"
},
"tileImage": "startpage/img/epg.png"

```

Listing 19: Link to EPG – Example

Name	Type	Notes
<code>tileImage</code>	URL	Both absolute and relative URLs are allowed.

Table A.8: Link to EPG parameters

Paused TV programmes

Displays single tile that corresponds to the first paused TV programme.

- When there is no paused TV program at all, this tile is not displayed.
- When there is at least one paused TV program, this tile either links to most recent program's detail or starts program playback directly. See parameter description below.
- When there are several paused TV programs, this tile can optionally link to a history page that lists all "Paused programmes".

Actions

keyOk Try to play or open detail of last paused programme.

REST API dependencies

`/subscription/content/get-visited`

Portal API dependencies

```

signals
  .saveStateOfVisited.addObserver
  .playbackEnded.removeObserver

```

```

"type": "visitedEpg",
"titles": {
  "eng": "Paused",
  "cze": "Rozkukáno"
},
"paused": true,
"play": true,
"moreToHistory": true

```

Listing 20: Paused TV programmes – Example

```

vod.run
  .history
dialogs.utils
  .entityDetails
tvApp
  .play

```

Name	Type	Notes
paused	boolean	Only paused programmes.
play	boolean	Start programme directly/open detail.
moreToHistory	boolean	Multiple items → go to history

Table A.9: Paused TV programmes parameters

Last recorded programmme

Displays single tile that corresponds the most recently recorded TV programme.

- When there is no recorded program at all, this tile can either disappear, or display a link to particular VOD movie, for example a video tutorial that introduces the PVR feature. See parameter description below.
- If there is exactly one recorded program, this tile either links to this program's detail or starts program playback directly. See parameter description below.
- In case of several recorded programs, this tile links to the PVR app which lists all recorded programs.

Actions

keyOk Try to play or open detail of last paused programme. Alternatively open history applet.

```

"type": "lastRecorded",
"titles": {
  "eng": "My records",
  "cze": "Moje nahrávky"
},
"fallbackEntityId": 7869,
"fallbackEntityPlay": true

```

Listing 21: Last recorded programme – Example

REST API dependencies

```

/server/vod/entity-detail
  for the fallback tile
/subscription/vod/pvr-programs
  for recordings, search API is delayed by elastic search

```

Portal API dependencies

```

pvrService
  .removeRecordingObserver
  .addRecordingObserver
playerApp
  .play
dialogs.utils.entityDetails
  .vodDetail
dialogs.utils.entityDetails
  .npvrDetail
vod.run
  .pvr

```

Name	Type	Notes
fallbackEntityId	number	Optional. Add alternative VOD movie.
fallbackEntityPlay	boolean	Optional. Start programme/open detail.

Table A.10: Last recorded programme parameters

Last rented VOD movie

Displays single tile that corresponds to the last rented VOD movie.

- When there is no recorded program at all, this tile is not displayed.
- When there is exactly one rented movie, this tile either links to this movie's detail or plays it.

- When there are several rented movies, this tile links to the VOD app which lists lists the rented movies.

```

"type": "lastOrderedVod",
"titles": {
  "eng": "Rented movies",
  "cze": "Zapůjčené filmy"
},
"params": {
  "providerCodes": [
    "code"
  ]
}

```

Listing 22: Last rented VOD movie – Example

keyOk Try to play or open detail of the movie. Alternatively open VOD browser applet with rented movies.

REST API dependencies

```

/subscription/search/search
/subscription/vod/purchased-entities

```

Portal API dependencies

```

purchaseService
  .addPurchaseObserver
  .removePurchaseObserver
dialogs.utils.entityDetails
  .vodDetail
vod.run
  .vod

```

Name	Type	Notes
params	JSON	
params.providerCodes	string[]	List of content provider codes.

Table A.11: Last rented VOD movie parameters

VOD movie

Displays single tile that corresponds to particular VOD movie. Either opens movie detail or starts movie playback when clicked.


```

    "type": "vodDetail",
    "entityId": 7869,
    "play": false

```

Listing 23: VOD movie – Example

Actions

keyOk Try to play or open detail of the specified movie.

REST API dependencies

/server/vod/entity-detail

Portal API dependencies

```

player.entity
    .VodEntity
player
    .WatchUtil
dialogs.utils.entityDetails
    .vodDetail

```

Name	Type	Notes
entityId	number	Identifies the VOD movie
play	boolean	Start programme/open detail.

Table A.12: VOD movie parameters

TV program

Displays single tile that corresponds to particular TV program. Either opens program detail or starts program playback when clicked. This tile differs from VOD movie tile in visual representation. Standard title is specified in JSON configuration. TV program name is displayed as overlay text over the tile image.

Actions

keyOk Try to play or open detail of the specified programme.

keyRed or **keyRecord** Record selected programme.

REST API dependencies

/server/vod/entity-detail

```

    "type": "epgDetail",
    "epgId": 10217742,
    "titles": {
    "eng": "Recommended programme",
    "cze": "Doporučujeme z \acs{TV}"
    },
    "play": true

```

Listing 24: TV program – Example

Portal API dependencies

```

epg.epgService
    .getProgramDetail
tvApp
    .play
dialogs.utils.entityDetails
    .epgDetail
pvr
    .RecordUtil
channelService
    .getChannel

```

Name	Type	Notes
epgId	number	Identifies the TV program.
play	boolean	Start programme/open detail.

Table A.13: TV program parameters

Banner

Displays ad banner according to banner API. Parameter *bannerType* can be one of *fullBanner* (460x60), *verticalBanner* (240x400), *squareButton* (125x125) or *button1*. (120x90).

Actions

keyOk Execute banner action.

Portal API dependencies

```

banners
configService
stbInfo
Ajax

```

```

"type": "banner",
"titles": {
  "eng": "Recommended",
  "cze": "Doporučujeme"
},
"bannerType": "fullBanner",
"bannerId": "startpagebanner"

```

Listing 25: Banner – Example

Name	Type	Notes
bannerType	string	One of types specified in description.
bannerId	string	Optional. More in API

Table A.14: Banner parameters

Link to series

Displays single tile corresponding to a series episode and links to the series section of the VOD app.

```

"type": "seriesLink",
"titles": {
  "eng": "Series",
  "cze": "Seriály"
},
"params": {
  "originalName": "Miranda",
  "providerCodes": [
    "nangu"
  ]
}

```

Listing 26: Link to series – Example

Actions

keyOk Open series episodes.

REST API dependencies

/subscription/search/search

Portal API dependencies

vod.run.vod

Name	Type	Notes
<code>params</code>	JSON	
<code>params.providerCodes</code>	string[]	List of content provider codes.

Table A.15: Link to series parameters

Link to episode detail

Displays single tile corresponding to a series episode and links to the series section of the VOD app.

```

"type": "seriesDetail",
"params": {
  "originalName": "The Ricky Gervais Show",
  "providerCodes": [
    "code"
  ]
}

```

Listing 27: Link to episode detail – Example

Actions

`keyOk` Open series listing.

REST API dependencies

`/subscription/search/search`

Portal API dependencies

`dialogs.utils.entityDetails`
`.vodDetail`

Name	Type	Notes
<code>params</code>	JSON	
<code>params.providerCodes</code>	string[]	List of content provider codes.

Table A.16: Link to episode detail parameters

Link to channel list

Single tile that links to the channel listing page in the TV application. *titles* property is optional. When *titles* are not specified, this tile displays the name of the channel list it links to.

```
"type": "channelsLink",
"titles": {
  "eng": "Channel list",
  "cze": "Seznam kanálů"
},
"tileImage": "startpage/img/channels.png"
```

Listing 28: Link to channel list – Example

Actions

keyOk Open channels listing.

Portal API dependencies

```
tvApp
  .addChannelListChangeObserver
  .removeChannelListChangeObserver
  .getCurrentChannelListKey
channelService
  .getChannelListName
tv.tvBrowser.init
  .switchToTvBrowser
```

Name	Type	Notes
tileImage	URL	Absolute or relative URL

Table A.17: Link to channel list parameters

Custom application (widget)

Displays single tile that links to ADK W3C widget or other external application.

Actions

keyOk Open custom application.

```
"type": "widget",
"tileImage": "http://example.com/chameleon.png",
"widgetId": "nangu-app"
```

Listing 29: Custom application – Example

Name	Type	Notes
tileImage	URL	Absolute or relative URL
widgetId	string	Widget ID or absolute URL

Table A.18: Custom application parameters

Portal API dependencies

`nStore.runWidget`

Link to TV/Radio player

Displays single tile that links to the TV/radio player application.

```
"type": "switchToSection",
"tileImage": "http://example.com/startpage/radio.png",
"section": "RADIO"
```

Listing 30: Link to TV/Radio player – Example

Actions

keyOk Switch to TV or Radio section.

Portal API dependencies

`tvApp.play`

Name	Type	Notes
tileImage	URL	Absolute or relative URL
section	string	<i>TV, RADIO</i> or <i>INTERNET_TV/RADIO</i>

Table A.19: Link to TV/Radio player parameters

Name	Type	Notes
title	string	Optional. Overrides movie title.
tileImage	URL	Overrides default image.
params	JSON	Lower.
params.providerCodes	string[]	Enumerates enabled providers.
params.sortField	string	Field used for sorting.
params.sortOrder	string	<i>ASC</i> or <i>DESC</i> .
params.genre	number[]	Enumerates enabled genres.

Table A.20: Link to VOD category parameters

Link to VOD category

Displays single tile that links to specific movie category in the VOD app. The movie category is specified implicitly by search filter, see parameter description below. Tile image corresponds to the first movie in the search result.

```

    "type": "vodCategoryLink",
    "titles": {
      "eng": "Sale",
      "cze": "Filmy v akci"
    },
    "tileImage": "http://example.com/override-movie-image.jpg",
    "params": {
      "providerCodes": ["code"],
      "sortField": "rating",
      "sortOrder": "DESC",
      "genre": "Documentary"
    }
  }

```

Listing 31: Link to VOD category – Example

Actions

keyOk Switch to TV or Radio section.

REST API dependencies

/subscription/search/search

Portal API dependencies

vod.run.vod

List of Figures

2.1	Application visual – screenshot	6
2.2	Embedded menu – screenshot	7
2.3	Tiles – screenshot	7
3.1	Frameworks scale comparison	12
4.1	React & Redux application structure	16
5.1	Plan for integration with the platform	20
5.2	Basic directory structure	21
5.3	Example of a folder in <code>reducers</code>	22
5.4	Menu enhancement	23
5.5	Search applications – screenshot	24
5.6	Reducers structure and workflow	25
5.7	Middleware usage	27
6.1	Layers tool at work	33
6.2	Application state in Redux developer tools	34
6.3	Saga events in Redux developer tools	34
6.4	React developer tools	35

List of Tables

5.1	Tile sources option parameters	31
6.1	Action event duration – 30 tiles, 100 observations	36
6.2	Action event duration – 100 tiles, 100 observations	36
A.1	Server configuration – required keys	40
A.2	Row parameters	42
A.3	Common tile parameters	43
A.4	Now on TV parameters	44
A.5	Recordings parameters	45
A.6	VOD movie category parameters	46
A.7	Categories parameters	46
A.8	Link to EPG parameters	48
A.9	Paused TV programmes parameters	49
A.10	Last recorded programme parameters	50
A.11	Last rented VOD movie parameters	51
A.12	VOD movie parameters	52
A.13	TV program parameters	53
A.14	Banner parameters	54
A.15	Link to series parameters	55
A.16	Link to episode detail parameters	55
A.17	Link to channel list parameters	56
A.18	Custom application parameters	57
A.19	Link to TV/Radio player parameters	57
A.20	Link to VOD category parameters	58

List of Abbreviations

ADK	Application development kit
API	Application programming interface
DOM	Document object model
DSL	Domain specific language
EPG	Electronic programme guide
ES6	ECMA-Script 6 language standard
HTML	Hyper-text markup language
EPG	Electronic programme guide
ID	Identifier
IP	Internet protocol
IPTV	Television over internet protocol
JSON	Javascript object notation
JSX	Declarative language used for React components
LESS	Stylesheet language based on CSS
MVVM	Model view view-model architecture
PC	Personal Computer
PVR	Programme video recording
OPVR	Programme video recording provided by operator
OTT	Over-the-top content
REST	Representational state transfer
RC	Remote controller
SP	Startpage application
SPA	Single page application
STB	Set-top box device
TV	Television
UI	User interface
URL	Uniform resource locator
VOD	Video on Demand
W3C	World Wide Web consortium
XML	Extensive markup language

Attachments

We include a zip archive with the complete project.

1. The Startpage project contains:
 - (a) Source code of the application
 - (b) Administration documentation in `.md` format
 - (c) `package.json` and build configuration files
 - (d) static images and `.less` style files

The project, as included, can be built with `node` and `npm`. To download dependencies, `npm install` has to be run from the project root folder. The application can then be served by a development server by running `npm start`. By using `npm run build:prod` an optimized, compressed production build is produced to the `dist` folder.

```
npm install          # install npm dependencies

npm start           # run application in development mode

npm run build:prod  # production build
```

The application requires to be run by the STB portal and to have its API available. The device also has to be provided with access to the application server and own a valid identity.