

Jarkko Tuunanen

Community-driven online service

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Media Engineering

Thesis

28 November 2014

Author(s) Title	Jarkko Tuunanen Community-driven online service
Number of Pages Date	61 pages + 2 appendices 28 November 2014
Degree	Bachelor of Engineering
Degree Program	Media Engineering
Specialization option	Digital Media
Instructor(s)	Olli Alm, Lecturer Jani Tarvainen, Competence Manager, Senior Developer
<p>The purpose of this thesis was to implement a community-driven online service for a fashion magazine. The aim was to create the number one fashion and beauty related online service in Finland.</p> <p>The online service consists of two platforms. The core of the service is built on top of Drupal content management system. Wordpress was selected as the platform for the blogs. The user interfaces for both were designed and implemented to be responsive.</p> <p>Layout management in Drupal is challenging due to the limitations in the core Region and Block system. Utilizing Panels and Chaos Tools Suite's Views Content panes modules for layout management and site building was found to be beneficial.</p> <p>The seeming ease-of-use of the Views module tends to result in difficulties formatting and maintaining content listings. Using entity-based Views listings proved to be practical in order to avoid common pitfalls related to maintainability, and processing and formatting the output.</p> <p>As a result of the comparison, utilizing a minimal base theme as a foundation for a customized theme turned out to be a good decision.</p> <p>The online service was launched on schedule the same day as the first issue of the paper magazine was published. The average unique visitors target was exceeded during the first 12 months since the launch. The implemented theme and the front-end build were seen satisfactory on the whole.</p>	
Keywords	Drupal, theme building, web, responsive web design, grid, layout management

Tekijä Otsikko	Jarkko Tuunanen Yhteisöllinen verkkopalvelu
Sivumäärä Aika	61 sivua + 2 liitettä 28.11.2014
Tutkinto	insinööri (AMK)
Koulutusohjelma	mediatekniikka
Suuntautumisvaihtoehto	digitaalinen media
Ohjaajat	lehtori Olli Alm Competence Manager, Senior Developer Jani Tarvainen
<p>Insinööriyössä tehtiin uuden yhteisöllisen muotilehden verkkopalvelu. Tavoitteena oli luoda Suomen suosituin muotiin ja kauneuteen keskittynyt verkkopalvelu.</p> <p>Verkkopalvelu rakennettiin kahdelle alustalle. Ydinpalvelu toteutettiin Drupal-sisällönhallintajärjestelmää käyttäen. Blogialustaksi valittiin Wordpress. Molempien käyttöliittymät suunniteltiin ja toteutettiin responsiivisiksi.</p> <p>Käyttöliittymien sommitelmien hallinnointi on haasteellista Drupalin ytimen Region- ja Block-järjestelmän rajoittuneisuuden vuoksi. Johtopäätöksenä todettiin Panels- ja Chaos Tools Suite Views Content panes -laajennusosien hyödyllisyys käyttöliittymien sommitelussa sekä yleisesti sivunrakennusprosessissa.</p> <p>Views-laajennusosan näennäinen helppokäyttöisyys johtaa helposti vaikeasti muotoiltaviin ja ylläpidettäviin sisältölistauksiin. Toteutuksen yhteydessä havaittiin entiteettiperusteisten Views-listauksien vahvuus tyyppillisten tiedon prosessointiin, muotoiluun ja ylläpidettävyyteen liittyvien sudenkuoppien välttämiseksi.</p> <p>Vertailun tuloksena minimaalisen pohjateeman käyttö koettiin onnistuneena valintana räätälöidyn teeman luonnissa.</p> <p>Verkkopalvelu julkaistiin aikataulussa yhdessä lehden ensimmäisen paperisen numeron kanssa. Keskimääräinen kävijämäärätavoite ylitettiin julkaisun jälkeisen ensimmäisen 12 kuukauden tarkastelujaksolla. Verkkopalvelun teeman ja käyttöliittymien toteutus todettiin kokonaisuutena onnistuneeksi.</p>	
Avainsanat	Drupal, teemoitus, verkkosivusto, responsiivinen verkkosuunnittelu, palsta, käyttöliittymien sommittelu

Contents

1	Introduction	1
2	Drupal & responsive web design	2
2.1	Drupal	2
2.1.1	Fundamental architecture concepts	4
2.1.2	Views module	11
2.2	Responsive web design	15
2.2.1	Media queries	16
2.2.2	Grid systems	16
3	Costume.fi	20
3.1	The process	20
3.2	Concept and general specification of the service	22
3.3	General architecture	23
3.4	User interface	25
3.5	Theme implementation	36
3.5.1	Base theme	36
3.5.2	High-level layout & grid system	38
3.5.3	Layout management	40
3.5.4	Views listings	46
4	Results	55
4.1	Usage data, user accounts and user-generated content	55
4.1.1	Usage data	55
4.1.2	User accounts	57
4.1.3	User-generated content	58
4.2	Analysis	60
5	Conclusions	61
	References	63
	Appendices	
	Appendix 1. Post node templates	
	Appendix 2. Layout 1 Panels layout template	



1 Introduction

Costume is a community-driven online service and a print magazine. The brand is built to be a community where members take an active role in the content creation for the online service and the print magazine.

The Costume brand landed in Finland in 2012. The online service, Costume.fi, was commissioned by Bonnier Publications. The concept, design, and development of the online service were done by Exove together with the client.

The aim was to create the number one fashion and beauty related online service in Finland. The target was to attract at least 55 000 unique visitors per week.

The process started with concept design stage in May 2012. The concept design stage was followed by design, technical implementation, and maintenance stages. Technical implementation was done mainly during July-August. The online service was launched the same day the first issue of the print magazine was published, August 22, 2012.

The online service consists of two platforms. The core of the service is built on top of Drupal whereas the blogs are served from a separate Wordpress multisite. The front-ends for both of the platforms were built using responsive web design methodology.

This thesis introduces the Drupal content management system, basic responsive web design techniques, and the overall process of how the online service was created. The focus in this thesis is on Drupal theme development and associated headaches. The key front-end related techniques of responsive web design are also discussed and demonstrated on a high level. The author of this thesis worked as a part of the development team being responsible for Drupal theme and front-end development. Other members of the team included back-end and CMS developers, designers, and project manager.

The client

The Costume brand in Finland was initiated and originally owned by Bonnier Publications. The brand was acquired by Aller Media in February 2014. Aller Media is a multimedia house specializing in entertainment, lifestyle and digital media concepts, and

social media business. The company employs approximately 250 employees in Finland.

2 Drupal & responsive web design

2.1 Drupal

Drupal is an open source software package originally developed in the year 2000. Drupal is primarily used as a web content management system for building web sites. The modular architecture of the system allows using it for a variety of use cases, for instance, to power a corporate web site, an e-commerce site, a communal online magazine, or as a web application framework. (Tomlinson & VanDyk 2010: 1)

Drupal is known as an extensible, modular and customizable content management system. The *Drupal core* is a stripped-down framework that only provides basic functionality – such as URL routing, basic content management, user management, and templating – that is used to support other parts of the system. The core can be extended with core modules, contributed modules (third-party add-ons), or with custom-made modules specific to the particular application. The core ships with approximately 50 modules, whereas the number of available contributed modules is counted in thousands. (Tomlinson & VanDyk 2010: 1-5)

The Drupal technology stack is based on popular technologies. Drupal can be run on any operating system that supports PHP (Hypertext Preprocessor). This includes effectively every major operating system such as Linux, Mac OS X, and Windows. Supported web servers include Apache, lighttpd, Nginx and Microsoft IIS. Drupal supports a variety of databases including MySQL, PostgreSQL, and SQLite, through a database abstraction layer that is used for interfacing Drupal with databases. Microsoft SQL Server and Oracle are also supported with the use of additional modules. Drupal is written in PHP. The user interface is constructed using HTML (HyperText Markup Language), CSS (Cascading Stylesheets), and JavaScript. (Tomlinson & VanDyk 2010: 2)

Drupal has become one of the most popular open-source content management systems in the world. At the time of writing, according to statistics provided by W3Techs, Drupal is the third most popular open-source content management system with a little

over 5% market share. The two more popular open-source content management systems are WordPress and Joomla with market shares of 60.4% and 8.1%, respectively. The technology statistics by W3Techs are based on the top ten million websites that are based on popularity rankings provided by Alexa using a three months average ranking. (Usage Statistics and Market Share of Content Management Systems for Websites, July 2014.)

Reasons for selecting Drupal as a content management tool vary. Like many other community open-source projects, Drupal's most valuable asset is the vibrant community behind it. A few examples of Drupal's strengths include the following [Hodgon 2013: 1–2]:

- Drupal is *free and open-source software* (FOSS). The Drupal core, and the core and contributed modules are released under GNU General Public License version 2.
- The Drupal community is an active group with thousands of members. The community is made up of individual volunteers and commercial organizations that are responsible for filing bugs, contributing patches and extensions, and maintaining documentation.
- The number of contributed modules is counted in thousands. There are a variety of widely tested, peer-reviewed modules available for download at drupal.org.
- Drupal utilizes commonly used technology stack familiar to most web developers.

Drupal positions itself as a tool for a wide range of applications. The flexibility of the system, however, comes at a cost. Drupal is often criticized for its complexity and for being overwhelming for users and developers. Many “Drupalisms” (Drupal-specific knowledge) have emerged over the years. Drupal 7 included greater abstraction and introduced a number of new APIs (Application Programming Interface). The increasing complexity of the system has resulted in steeper learning curve, making it less attractive for developers, and harder for organizations to find Drupal talent. (Buytaert 2013.)

Other common criticism toward Drupal include the following:

- The lack of separation between logic and presentation in the theme layer [Buytaert 2013].

- Deployment challenges [Buytaert 2013]. Drupal stores most of its configuration in database making the migration of site configuration between development and production environments troublesome and laborious.
- The lack of hierarchical page structuring. Creating tree-like structures is not natively supported in Drupal.
- Performance and scalability. Drupal is not known for its capability to process requests fast or to handle many requests simultaneously.
- Drupal does not include much of functionality out-of-the-box and therefore is not the best tool available for certain limited use cases, such as running a blog, forum, or wiki.

The current major version number of Drupal is 7. It was initially released in January 5, 2011. At the time of writing, the latest stable release of Drupal is 7.34 released in November 19, 2014.

The Drupal community is currently working on the version 8 of the Drupal core. Drupal 8 will undergo major architectural changes. The system will adapt to object-oriented programming, Symfony framework, and Twig templating layer, while preserving a lot of the basic concepts of Drupal such as Entities, Nodes, Fields, and Views. Some of the core initiatives in Drupal 8 include adding support for HTML5, making Drupal a mobile-friendly content management system, improvements in multilingual support, and adding Views module in the core [Drupal 8 Updates and How to Help]. The launch date for Drupal 8 has not been confirmed yet. (Buytaert 2013.)

2.1.1 Fundamental architecture concepts

When beginning to understand Drupal 7, there are some fundamental architecture concepts to be aware of: *entities*, *fields*, *modules*, *hooks*, *menus*, and user interface items such as *regions* and *blocks*.

Entities

Entities are an abstraction for data such as content or settings. A single entity can be thought of as an object, or an instance, of data. Entities are type-specific. The entity types included in the Drupal core are *nodes*, *comments*, *users*, *taxonomies*, and *files*. The Entity API introduced in the Drupal 7 core also allows the creation of custom entity

types. Examples of such custom entity types are products, customers, and groups. Implementations of entity types are called *bundles*. A bundle allows attaching fields to entities making them fieldable. Most entity types in the Drupal core are fieldable. (Hodgon 2013: 66-67.)

Node is the most notable entity type in the Drupal core. At its simplest, a node is a piece of content that has a title, and a collection of meta attributes that are stored in the database. A few examples of attributes common to all nodes include the following [Tomlinson & VanDyk 2010: 137-140]:

- *nid*: an unique ID number for the node
- *type*: the subtype of the node entity, for example a blog post
- *language*: the language of the node
- *created*: a Unix timestamp indicating when the node was created
- *changed*: a Unix timestamp indicating when the node was last modified
- *status*: an integer value indicating whether the node is "unpublished" or "published".

A node is a fieldable entity type that can be extended further into subtypes – known as *content types* or *node types*. A content type is a bundle consisting of zero or multiple fields that are attached to the node entity type. All content types inherit the data attributes from the base node, but can introduce new attributes relevant to the content type. (Tomlinson & VanDyk: 2010: 137-140.) Typical examples of a content type are an article, a blog post, or a flat page. Figure 1 demonstrates how content types are derived from the base node entity type. In the example only *blog_post* content type introduces data attributes of its own. The *blog_post* content type includes *image* and *tags* fields in addition to the inherited ones.

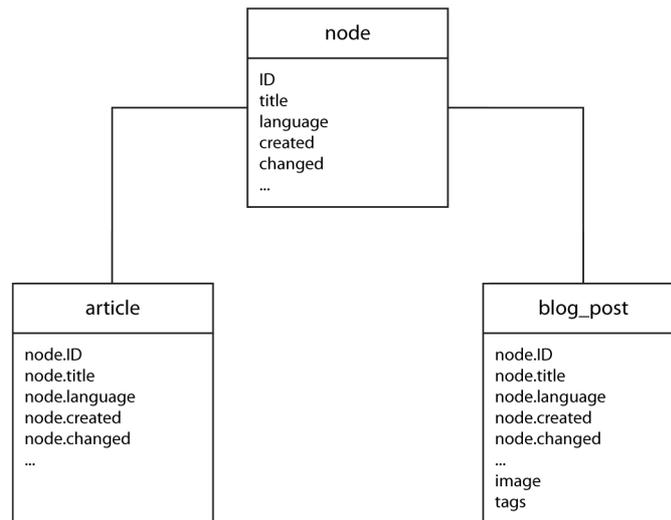


Figure 1: Content types are derived from the base node type but can also have data attributes of their own

Taxonomies are used for categorization of content in Drupal. Each of the taxonomies consists of organizational keywords – known as *taxonomy terms* – that are grouped together by *vocabularies*. A single vocabulary can be hierarchical or flat in nature, and can be associated with other entities. Typical examples for the use of taxonomies are categorizing content under major sections of a site, or a tagging system for blog entries. (Tomlinson & VanDyk 2010: 343.)

Every visitor in a Drupal site is associated with a *user* entity. A user entity can have one or multiple user roles. By default there are three roles in Drupal: *anonymous*, *authenticated*, and *administrator*. Additional user roles and permissions can be created at will. Visitors are considered as anonymous users until they sign up and log in to the system. Authenticated users can have one or multiple user roles, and optionally benefit from additional permissions. Authenticated users are also able to associate content they create with their account.

The user entity type is fieldable. By default it includes basic fields for holding account and personal information of the user, such as a username, an email address, and a password. Like most other entity types, the user entity type can be extended with additional fields.

All entities in Drupal are comprised of zero or multiple fields. A *field* is a reusable piece of content that can be attached to most entity types. Fields are used to organize the

data a single entity holds. Each field represents a primitive data type. Typical examples of a field include a title, an image, a description text, or a date. Fields can be shared among entity types, and the types and the number of fields per entity type are customizable.

Entity view modes

Entities can have *view modes*. A view mode defines what fields and settings for them are used when an entity is displayed under certain circumstances. In other words, entity view modes define a special representation for an entity depending on the context it is being displayed. For example, a node may have different set of fields and display settings when it is viewed on its own page (“Full” view mode, for example), and in listings (“Teaser” view mode, for example; Figure 2). Additional view modes for entities can be created using contributed modules such as *Entity view modes* or *Display Suite*, or they can be defined in a custom module. Entity types that are not directly displayed (e.g. internal-use entity types) do not need to have any view modes. (Hodgon 2013: 67–68.)

The screenshot shows the 'Article' entity configuration page in Drupal. At the top, there are tabs for 'EDIT', 'MANAGE FIELDS', and 'MANAGE DISPLAY'. Under 'MANAGE DISPLAY', there are buttons for 'Default', 'Full content', 'Teaser' (which is selected), and 'Headline'. Below this is a table with columns 'FIELD', 'LABEL', and 'FORMAT'. The table lists four fields: 'Image', 'Intro', 'Tags', and 'Body'. Each field has a dropdown for 'LABEL' (all set to '<Hidden>') and a dropdown for 'FORMAT'. The 'Image' field has a format of 'Image' and additional settings: 'Image style: Thumbnail (100x100)', 'Linked to content', and 'No class'. The 'Intro' field has a format of 'Plain text' and a class of 'intro'. The 'Tags' field has a format of 'Link' and 'No class'. The 'Body' field has a format of '<Hidden>' and 'No class'. A 'Save' button is at the bottom left.

FIELD	LABEL	FORMAT	
+ Image	<Hidden>	Image	Image style: Thumbnail (100x100) Linked to content No class
+ Intro	<Hidden>	Plain text	Class: intro
+ Tags	<Hidden>	Link	No class
Hidden			
+ Body	<Hidden>	<Hidden>	No class

Figure 2: Fields, their format and ordering, may be configured per-entity and per-view-mode basis.

Modules

Modules are used to extend or customize the functionality of a Drupal site. A module is a collection of functions and configuration files. It may also include templates and supporting code such as stylesheets and Javascript [Hodgon 2013: 11]. Generally, mod-

ules can be categorized under core or contributed modules, but developers are also able to create private custom modules specific to the application.

Core modules are shipped with Drupal and form the basic building blocks of a Drupal site. The foundation of Drupal is built upon core modules such as *Block*, *Comment*, *Entity*, *Menu*, *Node*, *System*, *Taxonomy*, and *User*. Core also includes optional modules for more specialized purposes, such as *Blog*, *Contact*, *Forum*, and *Poll*. Contributed modules are created by the community, and range from single task modules to complex solutions. Currently there are thousands of modules available for download for free at drupal.org website. Examples of popular contributed modules are *Views*, *Date*, and *Administration menu*. Developers are able to extend, or modify, Drupal functionality further by implementing a private custom module. Typically custom modules are developed for implementing specific project's needs.

Themes

Themes make up the look and feel of the site. Generally themes contain a collection of theme function overrides, template files, configuration files, and supporting files such as stylesheets, Javascript, and image files. (Hodgon 2013: 11–12.)

Theme functions and template files are used to generate the output – typically HTML markup – that is being sent to the browser. Theme functions and template files target a specific themable item, for example a node, or a field. Themes can override theme functions and template files introduced in modules and other themes. The presentation of most themable items can be customized in the theme. (Tomlinson & VanDyk 2010: 215–216.)

Similar to modules, themes can be categorized under core, contributed, and custom themes. *Core themes* are shipped with Drupal and are mainly used for administrative (*Bartik* and *Seven*) or demonstrational purposes (*Stark* and *Garland*). *Contributed themes* can be further categorized under “off-the-shelf” themes, and base themes (also known as starter themes). Off-the-shelf themes often include styling and graphics, and can be used “as-is”. In addition, some allow customization, typically through the administration interface. Base themes are specifically designed to provide a foundation on which to construct a new theme [Tomlinson & VanDyk 2010: 185–186]. Base themes are often opinionated and aimed to make theme development easier and faster by

providing a predefined set of regions, template files, front-end tools, and so on. Implementing a *custom theme* gives full control over the theming process. Custom themes often implement application-specific design that requires extensive customization.

Hooks

During the page loading process, Drupal goes through a series of internal events, known as *hooks*, allowing modules to hook into the flow of execution. Hooks provide a loosely coupled communication between modules allowing them to extend, or modify, functionality of a Drupal site. When a module wants to react to an event, it implements a particular hook after which it will be automatically called.

Regions and Blocks

Regions are sections of the page layout. Fundamentally, they are containers for hosting blocks. Regions are defined by the theme layer therefore each theme can have a different set of regions. The appearance and placement of regions in the layout is controllable via page templates and stylesheets.

Regions are global within the theme. Usually all regions are available on every page unless the theme includes multiple base page templates, each containing a different set of regions. Effectively, this means that all different layout structures must be taken into account while planning the region system for the site. Sites that do not have the same layout across the whole site usually end up having a complex region system, or the region system is found insufficient altogether [Hodgon 2013: 40].

Blocks are components of functionality or content placed in regions defined by the theme. A block can be thought of as a stand-alone container for hosting virtually any kind of content. A user login form, a list of latest comments, and main menu are examples of a block. Typically, blocks are implemented by core or contributed modules. Creating specialized blocks can be done in the block administration interface, with contributed modules such as Views, or programmatically using the Block API. (Shreves & Dunwood 2011: 123.)

Blocks are administrated using the Block Manager administration tool. Each block is assigned to a specific region in the theme, and can have additional configuration for controlling the visibility of the block. The block visibility defines whether a block is visi-

ble, or hidden, based on criteria specified by the administrator. The block visibility options are somewhat limited and only allow controlling the visibility based on the URL, content type, user roles and users, or with custom PHP script injected in the block visibility setting pane. (Shreves & Dunwood 2011: 128-130.)

Quite often, especially in medium to large websites, the restrictions imposed by regions and blocks are found unacceptable. In addition to the mentioned visibility option limitations, there are a number of other concerns related to blocks:

- A block can only have one instance. It is not possible to have blocks in multiple regions at the same time, or to have blocks with different settings on different pages.
- Blocks are unaware of the context they live in. Passing parameters to blocks is tricky and usually means scanning the page URL to pull the data out. Creating flexible and reusable components is close to impossible with blocks.
- Performance. Drupal generates blocks for all regions upon every page request unless the block visibility settings prevent it. This happens regardless of whether the region the block is placed inside is rendered in the page or not. (Shreves & Dunwood 2011: 629.)
- Managing blocks with the Block Manager administration tool is often overwhelming.

Ignoring these restrictions increases the risk of harder maintenance and lower performance. Fortunately, there are a number of workarounds and alternatives for managing blocks, and for replacing the region and block system altogether, provided by contributed modules:

- *Context* and *Multiblock* modules allow creating multiple instances of blocks. In addition, *Context* includes improved block visibility handling.
- *Delta* module allows creating multiple page layouts with distinct set of configuration settings (region, and other theme settings).
- *Panels* module allows creating customized layouts and includes a drag-and-drop editor for placing content within those layouts. It also includes *Chaos Tools Suite* module's system of context that enables content being aware of what is being displayed. Together with *Panels Everywhere*, *Panels* can even be used to replace the region and block system altogether.

Selecting the right tool and approach for overcoming the limitations of regions and blocks depends on the size and requirements of the site. Based on my experience, the core region and block system is hardly ever sufficient. The layout management strategy for Costume online service is explained in chapter 3.5.3 Layout management.

2.1.2 Views module

The contributed *Views* module is essentially a user interface for composing queries to pull information from database and presenting the queried data on screen. Views is designed for site builders and administrators allowing them to create, manage, and display lists of content, typically without writing any custom code. Views provides the ability to query entities such as nodes, comments, taxonomy terms, users, and custom entities; to filter and sort data; to relate one type of data to another; and to display data in a variety of formats. Typical use cases for Views are overriding default system pages, creating content listings, archive pages, image carousels etc. (Hodgon 2013: 39, 80–82; Working with Views.)

Views module has become the *de facto* way for creating formatted data listings in Drupal. It is the most popular contributed module with nearly 800 000 reported installs (as of July 2014) [Views]. Starting from Drupal 8, Views will be part of the Drupal core [Drupal 8 Updates and How to Help]. (Hodgon 2013: 39, 80–82; Working with Views.)

Views output formatting

Each list managed by Views is known as a *view*, and the output of a view is known as a *display*. A single view can have one to many displays. Displays are commonly exposed to other parts of Drupal in the form of a block or a page. Additionally, the display could be rendered as a RSS feed or Chaos Tools Suite Views *content pane*, for example. (Hodgon 2013: 82; Working with Views.)

The output format of an individual view display is handled by *Views style plugins*. Style plugins define how a view display should be styled, e.g. as a HTML list (,), an unformatted list (<div>), or a HTML table (<table>). *Row styles plugins* define how an individual record should be styled. Default row styles included are rendering the entire entity (entity-based output), or selected fields or attributes from entities (field-based output). (Hodgon 2013: 82.)

The rows (entities) in displays using *entity-based* output (Figure 3) go through the normal Drupal rendering process allowing the presentation to be configured in the theme layer. The row style plugin settings allow selecting a *view mode* each row (entity) should be rendered with. Available view modes depend on the type of the entity, and the *view modes* attached to it (see *Entity view modes* section on page 7).

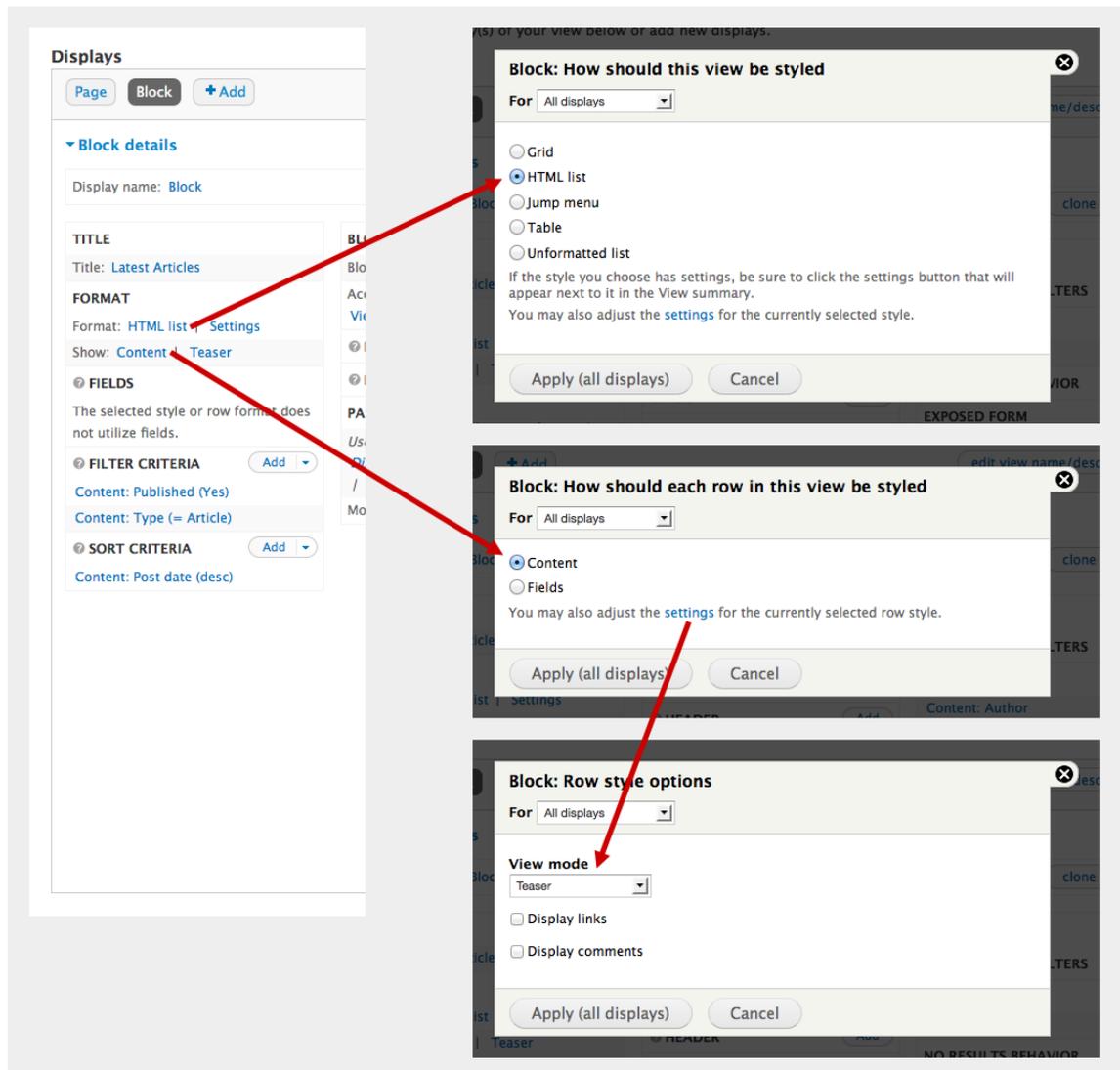


Figure 3: An example Views display configuration using HTML list as an output format, entity-based output (Content), and Teaser as row style formatter option.

Displays using *field-based* output (Figure 4) allow selecting arbitrary fields or attributes from queried entities. The order of the fields, and the display settings for each field, are configured in the Views administration interface. Each field is run through a field display handler that includes a variety of configuration options such as:

- Formatting the value of the field, for example prefixing node creation date with custom text (*Submitted on dd.mm.yyyy*).
- Formatting the appearance and functionality of the field, for example displaying an image using a specific image style, or making a link open in a new window.
- Selecting the wrapping element around the field, for example `<h1>` to `<h6>`, ``, `<div>`, etc.
- Controlling field label and its positioning.

The image shows the Drupal Views configuration interface for a block view. The main configuration panel on the left shows the following settings:

- Display name:** Block
- TITLE:** Latest Articles
- FORMAT:** HTML list (Settings)
- Show:** Fields (Settings)
- FIELDS:** Content: Post date, Content: Title
- FILTER CRITERIA:** Content: Published (Yes), Content: Type (= Article)
- SORT CRITERIA:** Content: Post date (desc)

Three modal windows are overlaid on the right, showing configuration options for the display:

- Block: How should this view be styled**
 - For: All displays
 - Selected: HTML list
 - Other options: Grid, Jump menu, Table, Unformatted list
- Block: How should each row in this view be styled**
 - For: All displays
 - Selected: Fields
 - Other option: Content
- Configure field: Content: Title**
 - For: All displays
 - Link this field to the original piece of content
 - STYLE SETTINGS:**
 - Customize field HTML
 - HTML element: H3
 - Create a CSS class
 - Customize label HTML
 - Customize field and label wrapper HTML
 - Add default classes

Figure 4: An example Views display configuration using HTML list as an output format, field-based output (Fields), and customized appearance settings for the title field.

Displays using field-based output provide a lot of flexibility to show the same content in different ways in different contexts. For instance, it would be rather simple to create two variations of “Latest articles” content listing in the following manner:

1. “Headline” listing showing only the meta-information (such as date and author), and the title of the node.
2. “Teaser” listing showing meta-information, title, intro text, and associated tags of the node.

Both of these listings could be configured entirely using Views administration interface. Field formatting, such as date format (i.e. *dd.mm.yyyy* vs. *mm/dd/yyyy*), and the generated HTML markup around the fields (``, `<div>`, etc.) could be configured for the *most part* using field style options.

In my experience, the “flexibility” and “ease-of-use” of field-based output typically generates more problems than they are worth, especially when it comes to building *sophisticated* content listings. The lack of separation of content and presentation, programmatic inaccessibility, the difficulty of authoring desired markup, and unsustainability, are all typical issues related to Views displays using field-based output (Views in general, actually). More specifically, common issues include the following:

- It is not possible to select and use different set of fields based on entity type in displays that query multiple types of entities. For instance, displaying category field for *Post* nodes, but not for *Blog entry* nodes in a shared listing.
- Adding conditional display logic (i.e. *show this field only if the value of another field equals to something*) is very difficult or impossible.
- Adding wrapper elements around multiple fields are supported only artificially (i.e. by using a “hack”).
- Views does not support HTML5 elements, such as `<article>`, `<header>`, and `<footer>`.
- Applying CSS classes to the row, or the field, based on a field value or entity property, is very difficult or impossible.
- Detailed layout configuration is embedded in Views. Presentational logic is stored in the database rather than in code (for example in theme files).
- Custom interface strings inserted in the configuration forms may not be translatable.

- Maintaining multiple views (each with multiple displays) is often laborious and error-prone due to high amount of configuration.

(There are ways to circumvent some of the mentioned problems by overriding Views templates, utilizing Views API and its hooks, or using contributed Views add-on modules such as *Semantic Views* and *Internationalization Views*, etc.)

Because of these reasons I chose to use field-based output cautiously in the content listing implementations for the Costume online service. Instead, I chose an approach where Views is mostly only responsible for the querying logic, leaving presentational logic to the theme layer as much as possible. See chapter 3.5.4 Views listings for details.

2.2 Responsive web design

Responsive web design is term used to describe a set of techniques for adapting a web user interface to the constraints of the browser window or device that renders it. The internet has moved beyond desktop-only use and is nowadays accessed using a number of devices, ranging from small-screen handhelds to widescreen devices – and anything in between. (Marcotte 2011: 6–9.). The aim of a responsive web design is to provide optimal viewing experience across a wide range of screen resolutions and devices the user interface is used. The three front-end related key technologies of building a responsive web user interface are [Marcotte 2011: 9]:

1. Media queries. Using media queries allows tailoring web user interfaces based on browsers' attributes and the capabilities of devices. Media queries are briefly introduced in chapter 2.2.1 Media queries.
2. Flexible, grid-based layout. Grid systems are a set of horizontal and vertical guidelines. They act as an invisible foundation helping to arrange content in a design in a consistent manner. Grid systems are explained in chapter 2.2.2 Grid systems, and the grid system used in Costume.fi is described in chapter 3.5.2 High-level layout.
3. Flexible images and media. A CSS technique to constrain images and media from displaying outside of their containing element. This topic is not discussed in this thesis.

2.2.1 Media queries

According to Media Queries specification "A *media query* consists of a *media type* and zero or more expressions that check for conditions of particular *media features*" [Media Queries]. Media queries are used to limit the scope of CSS styles based on browser's attributes, such as width, height, orientation, or pixel density. Utilizing media queries allows implementing presentations that are tailored to a specific range of output devices without making changes to the content itself.

In essence, media queries are questions that are asked from the browser. Media queries include two components: *media type*, and *media features*. Media types are used to classify browsers under a broad, media-specific category. Perhaps the most commonly used media types are *all*, *screen*, and *print*. "All", as the name implies, matches all types of browsers whereas "screen" matches only screen-based output devices, and "print" only printed output, respectively. Media features describe the characteristics of the output device. There are a number of features to test against but among the most commonly used are *width* (with *min-*, and *max-* prefixes), and *orientation*.

Styles enclosed in media queries are only applied if the browser matches the media type and the conditions specified in the query. If either of the criteria is not met, the browser will disregard the enclosed styles. Code example 1 demonstrates a simple media query that only targets screen mediums with a maximum viewport width of 639 pixels.

```
1. <link rel="stylesheet" type="text/css" href="narrow.css"  
2.     media="only screen and (max-width: 639px)"/>
```

Code example 1: An example how media queries can be used to limit the scope of styles

In Code example 1, the media query is defined in the "media" attribute of the <link> HTML element. If the browser conforms to the conditions specified in the media query, "narrow.css" file is used for presentation.

2.2.2 Grid systems

Joni Korpi [2012: 7] describes a grid system as "...a set of vertical and horizontal guides, which can be used to determine the sizes and positions of various elements in

a design in a consistent manner". Simply put, a grid system divides horizontal and vertical space into consistent units where content can be placed.

Grid systems can be categorized under *fluid-width* or *fixed-width* grid systems. In fluid-width grid systems, the width of an individual grid component, often called a *column* or *unit*, is expressed as a proportion relative to its container. As a consequence, fluid-width grids are adaptive by nature, making them particularly suitable for responsive layouts. Fluid-width grids can be "frozen" at certain widths to prevent columns from stretching too wide or too narrow, causing negative impact on text readability, for example. Figure 5 illustrates a fluid-width grid and a "frozen" fluid-width grid. Both grids are divided into three columns, each occupying 33 percent of the available horizontal space. In the "frozen" grid, the width of each column is prevented from stretching too wide by setting a maximum width to the containing element.

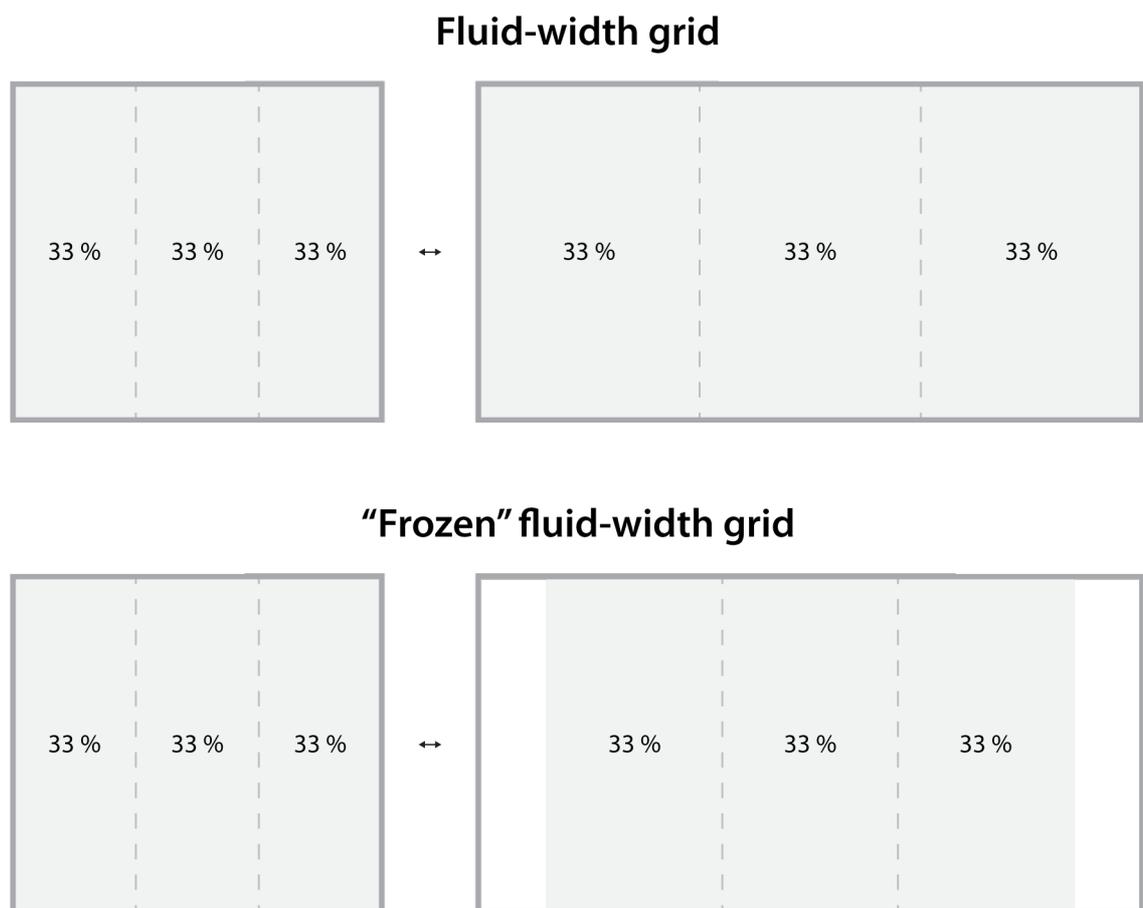


Figure 5: A fluid-width grid, and a "frozen" fluid-width grid

In fixed-width grid systems, the width of a column is expressed as pixels or ems. In pixel-based grid systems, the columns stay in proportion to the screen's actual physical pixels, unless overridden by the browser's viewport zooming setting [Korpi 2012: 10]. Figure 6 illustrates a pixel-based fixed-width grid where the column widths remain static as they are defined in pixels, rather than in proportional values.

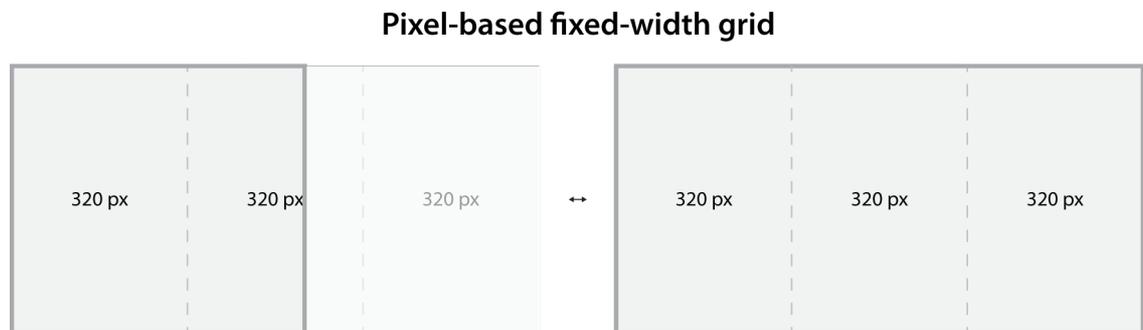


Figure 6: A pixel-based fixed-width grid

Fixed-width grids can be turned adaptive by adding or subtracting the number of columns as the viewport width changes. Generally, in adaptive fixed-width grids, the widths of individual columns do not change. Figure 7 illustrates an adaptive fixed-width grid where the column count varies based on the viewport width.

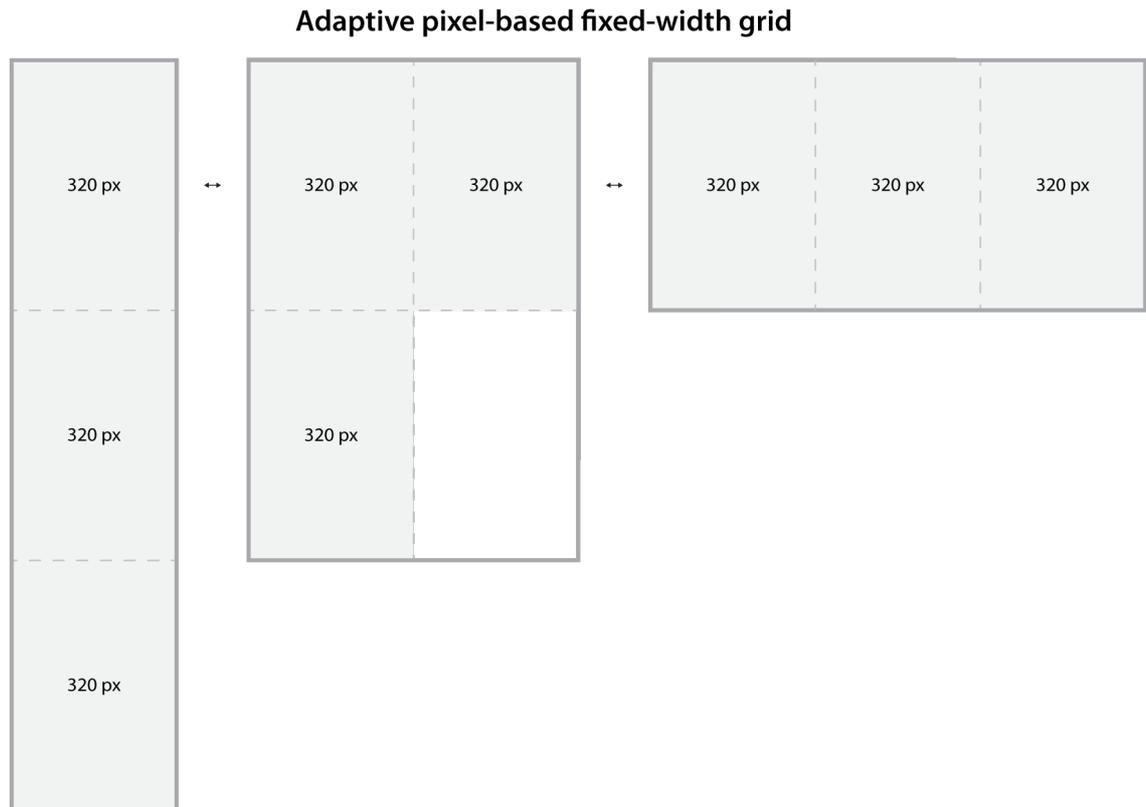


Figure 7: An adaptive fixed-width grid. Columns are added or subtracted as the viewport width changes

In em-based grids, the column width is determined by the font size of the element. In CSS, one "em" equals the element's font size in pixels, and it may be different for each element. By changing the font size, em-based grids can be scaled up and down, making the grid "elastic". An example of an "elastic" em-based fixed-width grid is demonstrated in Figure 8.



Figure 8: An "elastic" em-based fixed-width grid

Grid systems can be used as an invisible foundation to achieve visual cohesion in design. As grid systems are restrictive by nature, they enforce consistency throughout the designs. Generally, grid systems are used for structuring the high-level layout of the page. The sole purpose of a grid system is to abstract layout information away from the components, making construction of, and maintaining, page layouts faster and easier.

A grid system can be bundled into a *grid framework*. A grid framework is a reusable abstraction of code that includes a collection of CSS classes and rules that makes building grid-based layouts easier. There are a number of grid frameworks available with different kinds of approaches for both traditional and adaptive design.

3 Costume.fi

The previous chapter introduced the Drupal content management system, some of its architecture concepts and characteristics, and the Views module. In addition, the basics of responsive web design techniques were covered at the theory level.

In this chapter, we will go through the overall process of how Costume.fi online service was built – from concept design to launch. As we progress through the chapter, the architecture and implementation details of the online service are explained. The focus is on the technical implementation of the Drupal theme and its user interface. I will present some of the technical design decisions that were made and underline the reasoning behind them.

3.1 The process

The process had four major stages:

- Concept design
- Design
- Technical implementation
- Maintenance

Concept design was the initial stage of the process where the goals and objectives for the online service were defined. The concept is based on Bonnier's business strategy

for Costume.fi and describes the purpose of the online service. It defines the boundaries on what should be taken into account when developing the structure, functionalities, content and visuals for the online service. Exove Design did the concept together with Bonnier staff. Chapter 3.2 briefly describes the concept of the online service.

The design stage can be divided further into two categories: *Visual design* and *Architecture design*. The overall look and feel of the site is created in the visual design stage. Based on the brand and business strategy, visual design determines the typography, colors, page grid, layout and overall graphic design. The visual design was done partly in parallel with the technical implementation stage.

In the architecture design stage, the technical design of the entire system is planned. Based on the business needs, the methods – for example base systems – to reach the requirements are chosen. The high-level architecture design was already made before and during the concept design stage. For instance, based on business strategy, Drupal was chosen as the main platform for the online service already during the proposal stage. The general architecture of the Costume.fi is explained in more detail in chapter 3.3.

The actual development of the system is done in the technical implementation stage. The specification made during the concept design and design stages was used to implement necessary modules and components of the system. During the technical implementation stage, the visual design was also finalized based on the received feedback, after which the user interface was constructed based on the visuals. The implementation of the user interface is explained in chapters 3.4 and 3.5. The technical implementation stage also enclosed testing of the service. The functionalities, and user interface were tested as they were built.

After the technical implementation was done, the online service was launched, and the maintenance stage begun. During the maintenance stage, the performance and availability of the system was monitored. Tracking statistics and gathering user feedback was also an important part of the maintenance stage. Additional small-scale development was also done.

3.2 Concept and general specification of the service

The Costume brand in Finland consists of a print magazine, an online service, a mobile application, and a party concept. The brand focuses solely on fashion and beauty. The brand is built to be a community where members take an active role by participating in content creation for the online service and the magazine.

The core target audience is people between the ages of 18 and 29. A typical member of the target audience is a young female with an interest toward fashion, beauty, shopping, and blogging. The members of the target audience can be considered experienced internet users as a majority of them are blog devotees.

The concept design for Costume.fi was based on four principles: equality, visuality, community, and brand.

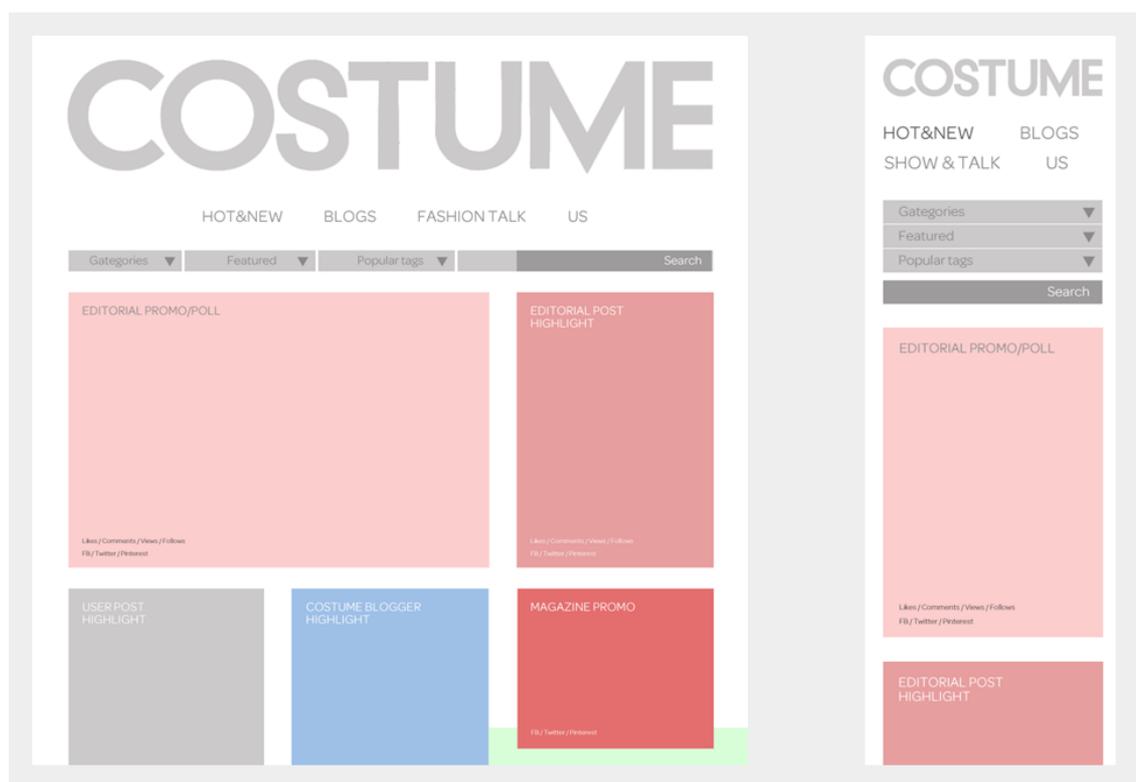


Figure 9: Early wireframe of the online service home page.

The core of the Costume.fi online service is the Fashion talk section – a forum where community members among the Costume staff can publish, comment, "love", and share fashion and beauty related content. Individual piece of content, a post, can in-

clude pictures, videos and text, and can be categorized under a variety of predefined categories. Other major sections of the online service are the home page, blogs, competitions, info pages, and community and user profile pages.

Available participatory functionality for users is based on the role of the user. Unregistered, anonymous users can access all major sections of the site and are allowed to read the content and comments. Unregistered users are also allowed to participate in discussions and competitions, and "love" and share the content created by others. Users will be allowed to sign up to the online service by creating a local account, or by connecting their Facebook account with the service. A user can create, modify and share their own profile and browse profiles of other members. Registered users can participate in content creation by publishing posts to the Fashion talk forum.

The varying use cases and environments of use will be taken into account in the implementation of the online service. The online service will be made accessible to a wide range of devices including handheld devices, laptops and desktop computers. Navigating, and interaction with the interface will be made possible using touch-based devices, such as tablets and smart phones. In order to serve the optimal viewing experience regardless of the available screen estate of the device or the rendering capabilities of its browser, the user interface will be built according to responsive design and graceful degradation methodologies. Graceful degradation is a front-end strategy where the user interface is built modern browsers in mind but where it remains functional in less feature-rich browsers.

3.3 General architecture

The Costume.fi online service was built by a team of developers, designers, and project managers together with the client. I was working primarily on the core service and was responsible for the Drupal theme and front-end development.

Based on the specification, the key functionalities of the online service include the following:

- User registration with an option for connecting a Facebook account with the service

- Fashion talk forum where members of the Costume community and staff can publish content
- "Loving", sharing and commenting of content
- Community section with individual user profile pages
- Blogging tools for selected members of the community
- Blog imports from external sources
- Competitions
- Integration between the core service and the mobile application
- Responsive user interface

The online service consists of two main platforms: the core service and the blogs. The core service was built using Drupal 7 content management system. Wordpress was set up as the main platform for the blogs. Integrations include blog imports from Wordpress to Drupal, integration between a separate mobile application and the core service, and Facebook integration for the core service. The mobile application was developed by another vendor. Figure 10 illustrates the general architecture of the service.

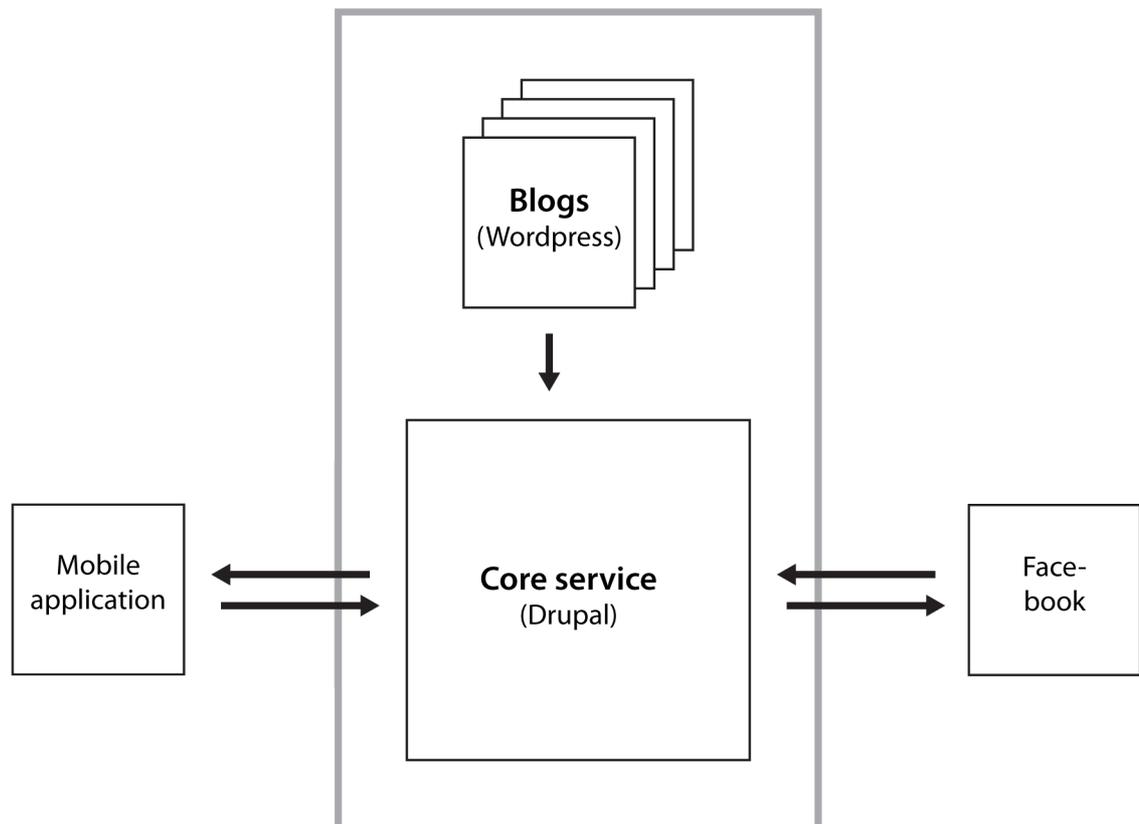


Figure 10: General architecture of the Costume online service

The core service and the blogs are served from a virtual private server. The server is equipped with two virtual central processing units and four gigabytes of memory. The server technology stack includes Nginx web server with FastCGI and PHP-FPM (FastCGI Process Manager for PHP).

Solutions for a majority of the key functionalities of the core service were available either in the Drupal 7 core or as contributed modules to the core. A few examples of the contributed modules used in the site include:

- *Facebook Connect* is used for connecting a user's Facebook account with the Drupal account
- *Radioactivity* is used for counting the number of views of individual posts and user profiles
- *Rate* with *VotingAPI* are used for tracking "loves" of individual posts and comments
- *Webform* is used for creating, reviewing, and storing custom competition forms
- *Media*, *Media YouTube* and *CKEditor* were used for providing a single-button solution for handling image and video uploading inside posts

Some of the functionalities were specific to the application, and thus required implementing private custom modules. For example, a custom-made REST API was developed to handle the integration between the core service and the mobile application. In addition, a number of the functionalities provided by core and contributed modules were modified and extended further with custom modules. For instance, the solution for importing Wordpress blogs and external blogs to the core service was based on *Feeds* module, but the import functionality was customized utilizing Feeds module hooks. Similarly, the Facebook integration functionality provided by Facebook Connect module was heavily customized to satisfy the project's needs.

3.4 User interface

The user interface for the Costume online service was built according to responsive design methodology. Responsive user interfaces often include multiple layout designs, each targeted to a set of devices, such as ones for mobile, tablet, and desktop. One fundamental choice when designing a responsive user interface is to decide which layout to design first. The two most prominent approaches are either designing first for the most constrained environment – in terms of screen estate and performance, for exam-

ple – or the least constrained environment. Starting with the most constrained, typically also the narrowest, design is known as *mobile first* approach. Starting with the full-size design and working the way down from there is known as *desktop first* design.

The designs for the Costume online service were done using the desktop first approach. The organization of content, content input, and page layouts were primarily designed with desktop users in mind. The site includes three major layouts: desktop (*normal*), tablet (*medium*), and mobile (*narrow*). The tablet and mobile layouts are ports of the full-size desktop version.

During the concept design stage, the site architecture of the online service was defined. The site architecture was prototyped and demonstrated using wireframes. In website design, a wireframe is a skeletal framework used to depict the page layout, including interface elements and navigational items, and how they work together [Website wireframe]. Wireframes were created for every major section of the online service, including the home page; Fashion talk, Blogs, Competitions, Us, and Info section. After the wireframes were finalized and approved, they were used as a visual guide for user interface design, and development. During the visual design stage, the wireframes were turned into comprehensive layouts, demonstrating what the final user interface would look like. The wireframes and their respective page layouts and implemented user interfaces for the home page and Fashion talk section front page are shown in Figure 12 and Figure 15.

The layouts contain *common page elements* and *page specific elements*. Common page elements are used throughout the site. They include navigational items, brand elements, search form, and advertisements. Common page elements are shown in Figure 11 and described in Table 1. Page specific elements may vary from page to page. Elements for the home page and Fashion talk section front page are shown in Figures 12–13 and Figures 14–15, and described in Table 2 and Table 3.

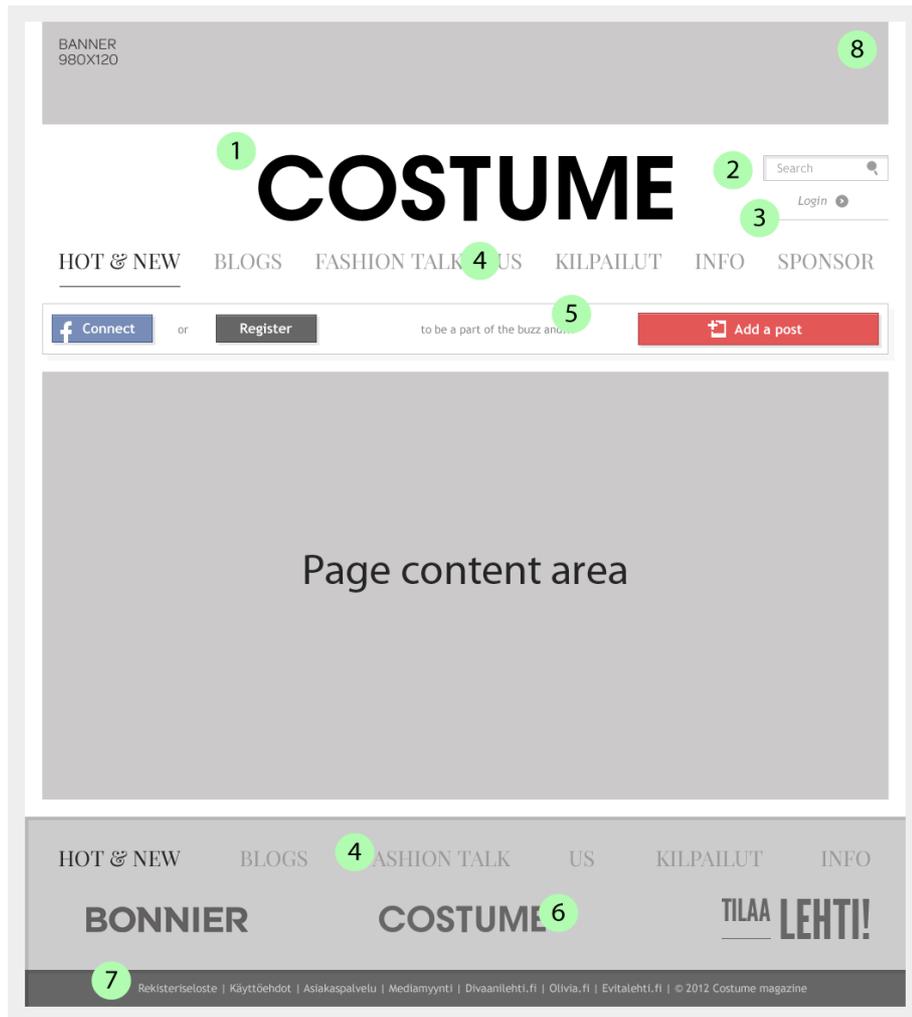


Figure 11: Common page elements. See Table 1 for explanations.

Table 1: Common page elements. See Figure 11.

#	Element	Description
1	Logo	Site logo for brand identification.
2	Search form	Functionality allowing users to search for specific content.
3	Login / logout link	Links for logging in or out, depending on the state of user's current authentication.
4	Main menu	Holds links to all major sections of the site.
5	Action bar	Contains buttons for Facebook login, registration, and adding posts.
6	Brand menu	Brand related logos and links.
7	"Fine-print" menu	Contains supplementary links, links to legal statements and other services, and copyright statement.
8	Advertisement banner	Promotional material.

The home page (Figure 12 and Figure 13) includes various lift-up and promotional elements, giving an overview of what is new in the site. Majority of the content is user generated. Editorial content is only prioritized in the *Fashion Talk* (or *Small Talk*) element.



Figure 12: Wireframe, page layout, and implemented user interface of home page (I / II). See Table 2 for explanations.

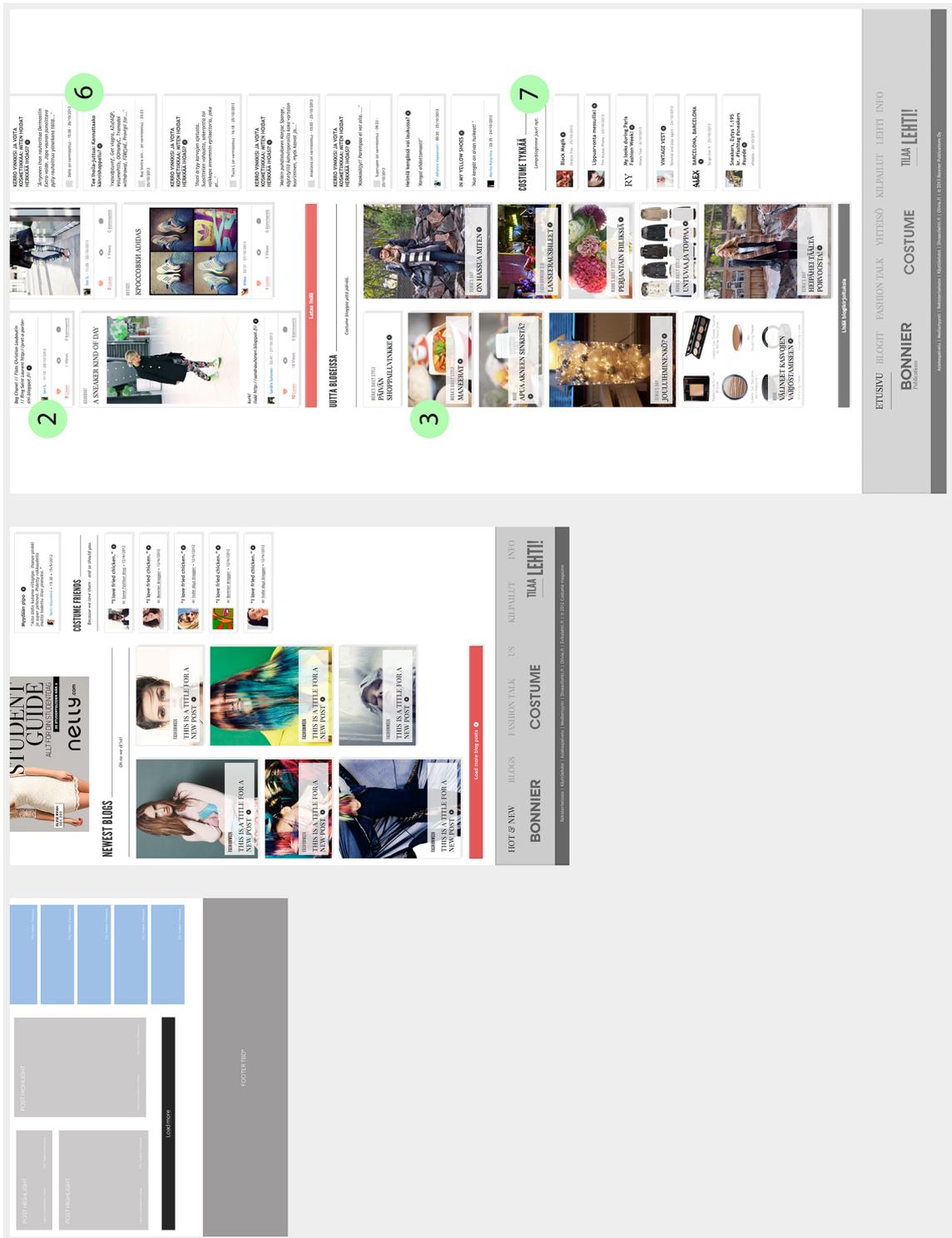


Figure 13: Wireframe, page layout, and implemented user interface of home page (II / II). See Table 2 for explanations.

Table 2: Home page elements. See Figures 12 and 13.

#	Element	Description
1	Psst...! / Sizzling hot	Post lift-ups displayed in random order.
2	Fashion talk posts	Latest posts from all Fashion talk. Editorial content is prioritized (two top-most items).
3	Blog posts	Latest blog entries created by Costume bloggers.
4	Bloggers	Links to Costume bloggers' blogs.
5	Active users	A selection of authenticated users displayed in random order.
6	Recent discussion	List of most recent comments.
7	Blog lift-ups	Latest blog posts from external blogs.
8	Promotion	Promotional banners.

The focus in the Fashion talk section front page (Figure 14 and Figure 15) is on the fashion and beauty related posts produced by the community and editors. The page includes options to filter the posts by category and tags. Other user interface elements include promotional banners, list of active users, and recent discussion.

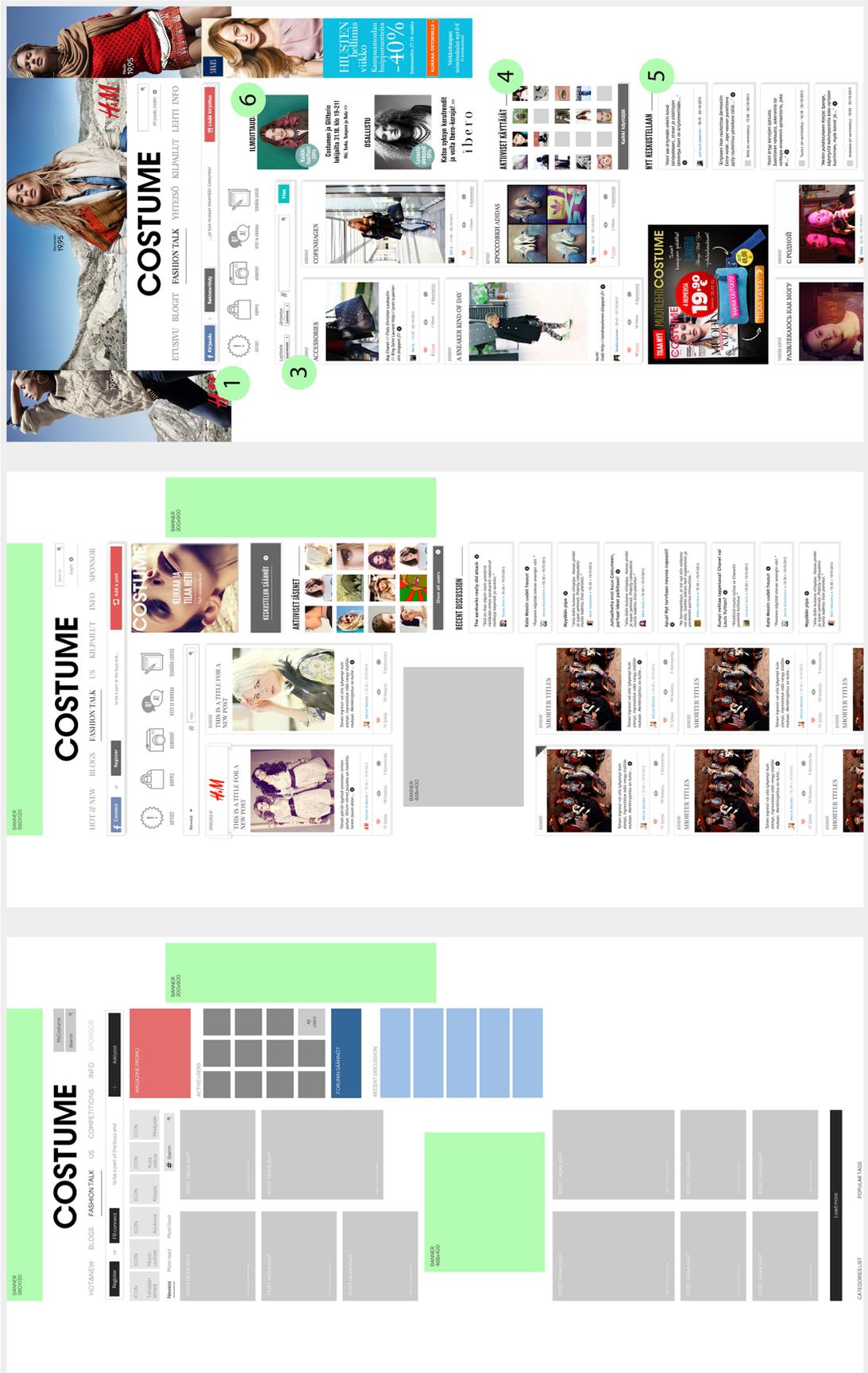


Figure 14: Wireframe, page layout, and implemented user interface of Fashion talk section front page (I / II). See Table 3 for explanations.

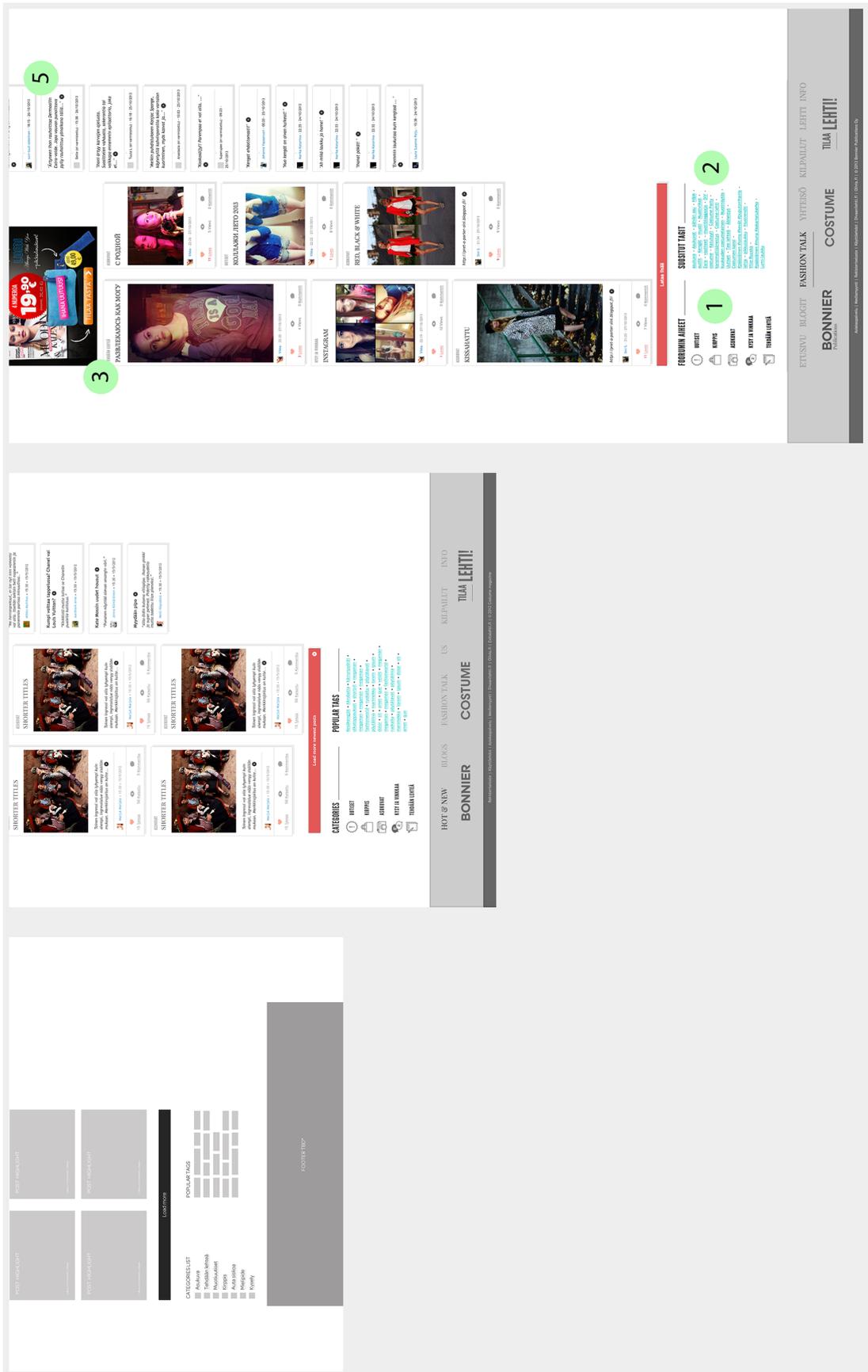


Figure 15: Wireframe, page layout, and implemented user interface of Fashion talk section front page (II / II). See Table 3 for explanations.

Table 3: Fashion talk section front page elements. See Figures 14 and 15.

#	Element	Description
1	Fashion talk categories	Links to individual Fashion talk category archive listings.
2	Fashion talk tags	Links to most popular tag archive listings.
3	Fashion talk posts	Posts ordered by their date in descending order. Component includes options to order posts by their date, and “love” count, and to filter posts by tags.
4	Active users	A selection of authenticated users displayed in random order.
5	Recent discussion	List of most recent comments.
6	Promotion	Promotional banners.

In responsive user interfaces, majority of the page content is used across all page layouts. Changing the appearance of the user interface, and the elements inside it, is done primarily with stylesheets. Designing and implementing responsive user interface elements requires attention, as the elements need to be flexible enough to adapt to multiple page layouts. In addition, navigating and interaction with the user interface should be made possible using touch devices. The logic behind the responsiveness and the functionality of the user interface was a joint effort between the designer and me.

Figure 16 demonstrates the differences between the desktop, tablet, and mobile page layouts of the user profile page. Examples of responsive design techniques used in the page include:

- Elements collapse on top of each other as the viewport width decreases: search box and login/logout link fall under the logo; secondary content (active users list) falls under the primary content; user name and information fall under the profile picture; posts are displayed side-by-side only when there is enough horizontal space.
- Main menu turns from a horizontal link list to a vertical link list. In addition, the vertical version of the menu includes an extra “Navigation” link that toggles the visibility of the main menu.
- Advertisement banners and wallpaper are only visible in the desktop layout.
- Posts are fluid elements. Their width and height varies depending on the available space.
- Parts of the action bar content are hidden in narrower viewports.

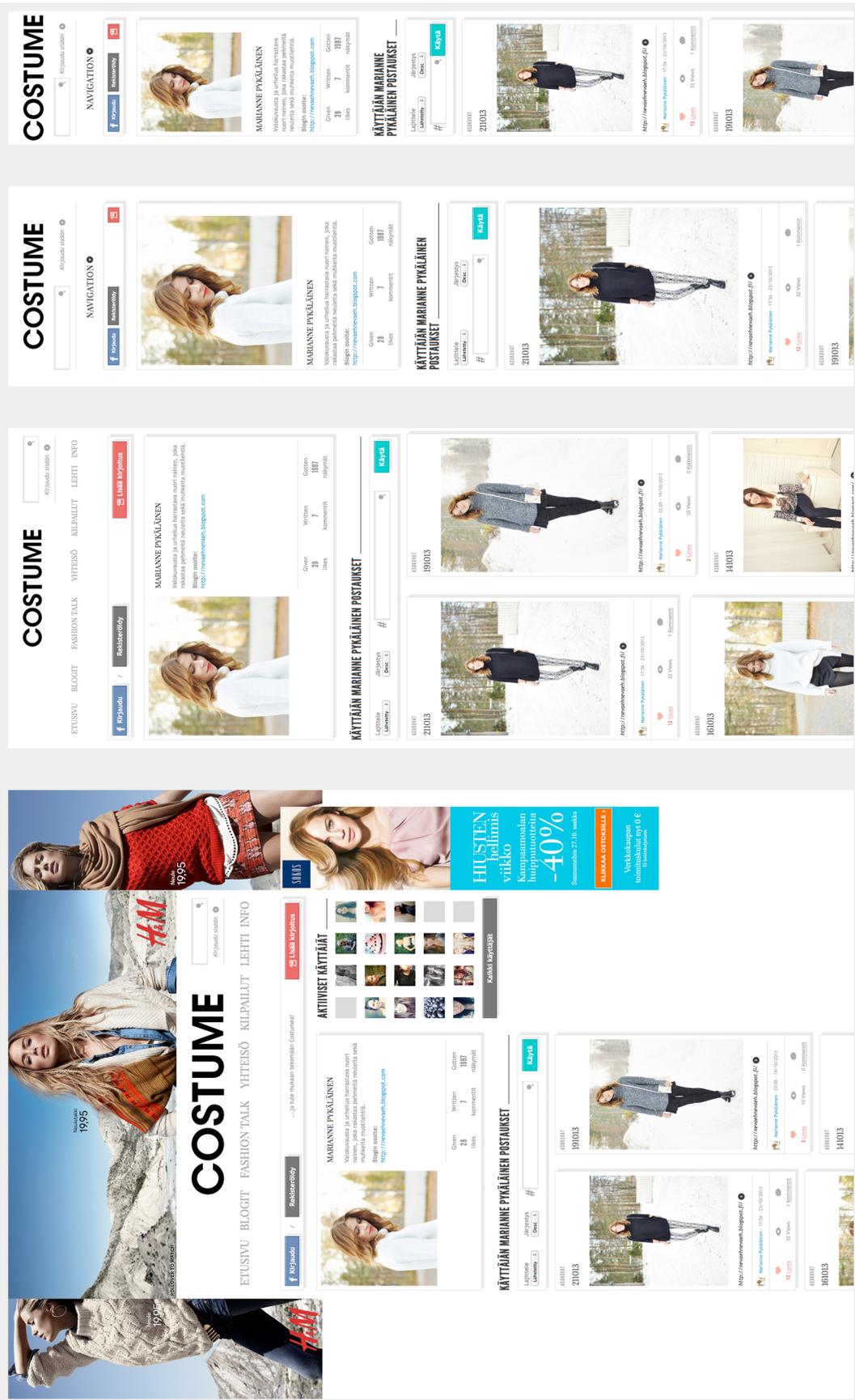


Figure 16: User profile page viewed in 1460, 768, 480, and 320 pixel wide viewports.

The *Action bar* shown in Figure 17 is an example of a fluid element, and demonstrates how user interface elements can respond to the changes in the viewport width. The Action bar is displayed to the user as a floating layer in the browser window. The bar has two different views depending on the user role. For anonymous users, the bar holds the following items:

- Registration/logging in with Facebook account
- Registration without Facebook account
- Add post button. Clicking the button redirects to the registration page.

For registered and logged in users, the action bar holds a different set of items, including:

- Profile picture, or a placeholder picture
- User name linked to user profile page
- Add post button

The space between the Register and Add post button is filled with a "filler" text, encouraging users to register and start producing content. The actual text varies depending on the user role.

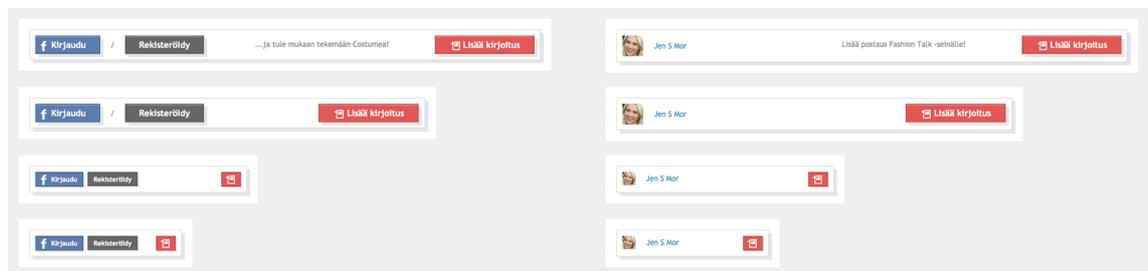


Figure 17: Action bar

The functionality of the Action bar remains the same regardless of the viewport width. As the viewport width decreases, the appearance of the Action bar is changed in the following manner:

- The size of the buttons and the profile picture is decreased
- Filler texts are removed
- Add post button text is removed

3.5 Theme implementation

The look and feel of a Drupal site is done primarily in the theme layer. Themes are constructed using theme functions, templates, stylesheets, scripts, assets, and other supporting files [Hodgon 2013: 11–12].

Theme implementation in Drupal goes hand in hand with the Drupal build itself. General Drupal architecture, and content model of the site generally affect the theme implementation. Modules and themes – and the combination of them – govern the high-level presentation method. As there are many different ways to implement a Drupal site and its theme, it is advisable to plan ahead and decide the practices and tools to use.

The starting point for the Costume theme implementation was to create a maintainable and easily extendable theme. In order to address many of the problems I had run across in other Drupal sites and themes, the following Drupal theming and front-end related design decisions were made:

- Panels, and Panels layouts over standard regions
- Chaos Tools Suite's Views Content panes over standard blocks
- Entity-based Views listings (using custom entity view modes) over field-based Views listings
- HTML5 templates
- Sass (Syntactically Awesome Stylesheets) over vanilla CSS
- Base theme supporting all of the previous, or no base theme at all

This chapter explains parts of the theme implementation for Costume highlighting solutions for some shortcomings of Drupal and its contributed modules. The focus is on Drupal theming related concepts: base themes, layout management, and Views listings.

3.5.1 Base theme

Perhaps one of the most fundamental choices when starting theme building in Drupal is to choose between on utilizing a *base theme* (or *starter theme*) as a foundation, or building the theme from scratch. There are a variety of base themes available for download at drupal.org website. Base themes are often opinionated and built around

specific tools, libraries, and languages. The basic concepts and features of base themes vary. Popular concepts include HTML5 templates, responsive design, grid systems, accessibility, and customization of the theme through user interface. [Starter themes].

While it is possible to modify, override, or disregard some of the basic concepts of base themes, having to work around, or against, a base theme is often undesirable. Selecting a base theme that has its own layout system, for example, would be nonsensical if the plan is to use Panels for layout management. Similarly, a base theme that has a lot of layout-driven styling that does not fit into the design is likely to only get in the way.

Deciding on whether to utilize a base theme – and the selection of a base theme – can be complicated. The site architecture, content model, design, front-end tools and libraries, and developer experience are all things to take into consideration.

Before starting the theme development for Costume, I researched a few base themes and their suitability for the site. At the time of implementation, many members of the Drupal community vouched for *Omega* and *Adaptive theme*. The 3.x branch (latest at the time) of the Omega theme was built around the concept of zones (groups of regions), layout configuration using Context and Delta modules, HTML5 templates, 960.gs CSS grid framework, and configuration through user interface [Omega]. *Adaptive theme*, another popular base theme, had its own layout system (Gpanels), two fixed sidebars, HTML5 templates, and also similar “point-and-click” type of configuration [Adaptive theme].

Omega and Adaptive theme both seemed too opinionated, and contained a lot of features uncalled-for. For instance, I saw no added value of being able to configure layout settings, styles, and media queries using the administration interface. In addition, the unique built-in layout system in Omega would have likely made switching out from the theme difficult, possibly resulting in theme lock-in.

I set out to find a minimal solution that would convert the core Drupal templates to HTML5, while preserving the standard Drupal theming concepts. I came across *Boron* theme that was built this philosophy in mind [Boron].

The main features, with emphasis on the suitability as a base theme for the Costume site, of Omega, Adaptive theme, and Boron are shown in Table 4 [Omega, Adaptive theme, Boron].

Table 4: Comparison of main features of Omega, Adaptive theme, and Boron base themes.

	Omega (7.x-3.x branch)	Adaptive theme (7.x-2.x branch)	Boron (7.x-1.x branch)
HTML5 templates	Yes	Yes	Yes
Grid framework / system	960.gs	Conventional grid	No
Responsive	Yes	Yes	No
Built-in layout system	Zones (groups of regions)	Gpanels (optional)	No
Built-in Panels support	No	Yes	No
Opinionated layout structure	Yes	Yes	No
“Point-and-click” theme configuration	Yes	Yes	No
Layout-driven CSS	Yes	Yes	Minimal

Instead of trying to modify the opinionated Omega or Adaptive themes to meet the design decisions and requirements, I ended up selecting Boron as the base theme for the site. Using a minimal base theme gave the freedom of selecting front-end tools and libraries of choice, and to plan the front-end architecture accordingly.

3.5.2 High-level layout & grid system

The high-level layout was built using a “frozen” fluid-width grid. The grid consists of one to three fluid columns separated by fixed-width gutters. The number of columns is determined by the viewport width. The narrower the viewport, the fewer the columns. Figure 18 illustrates the initial grid system of the Costume online service.

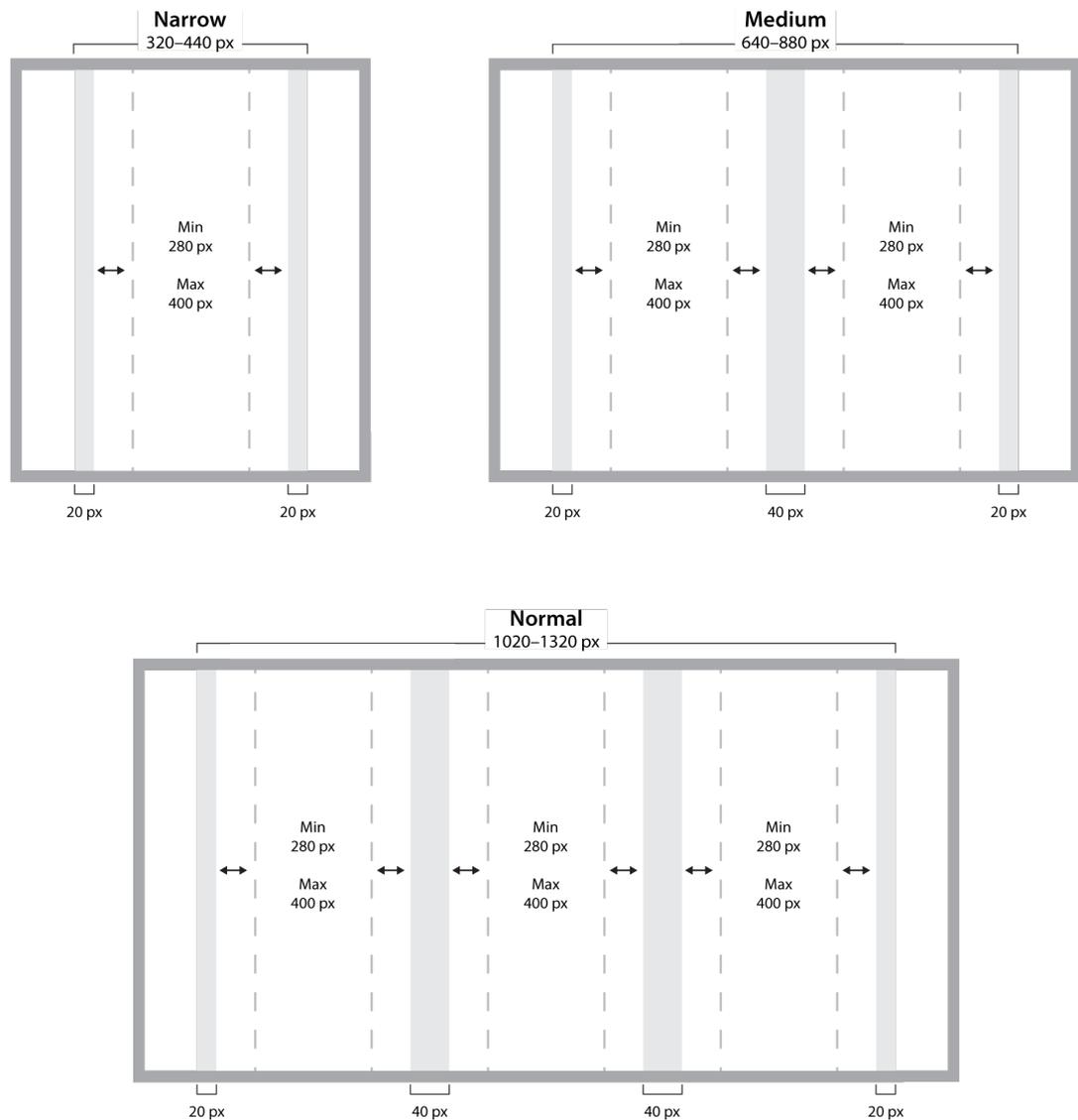


Figure 18: Initial “frozen” fluid-width grid system of the Costume site.

The high-level layout has three major breakpoints: *narrow*, *medium* and *normal*. In each layout, the gutter width remains at 40 pixels while the widths of the columns are constantly in flux. Each layout has one restriction: the overall width of an individual column must be between 280 and 400 pixels. Given the restriction the narrow layout is active when the viewport width is less than 640 pixels, the two-column medium layout is active between 640 and 1019 pixels, and the three-column normal layout activates at 1020 pixel viewport width. The maximum width of the columns was handled by setting a maximum width to the containing element of the grid, not the columns themselves.

The requirements for the grid changed during the implementation phase. An additional “wallpaper” advertisement was added to the three-column layout. The wallpaper is dis-

played in the background behind the main content. The allocated area for content on top of the wallpaper was narrower (980 pixels) than the original maximum width of the grid (1320px). In order to address this issue, the page content was limited to span only 980 pixels wide even if the viewport width was significantly higher. As the columns were fluid, the change did not require any changes to the grid system itself. However, the specification for column minimum and maximum widths in three-column layout, and the starting breakpoint of the layout were affected (shown in Figure 19). After the change, the three-column layout activated already at 1000 pixels. The column widths effectively got frozen, and were not able to span any narrower or wider than approximately 286 pixels.

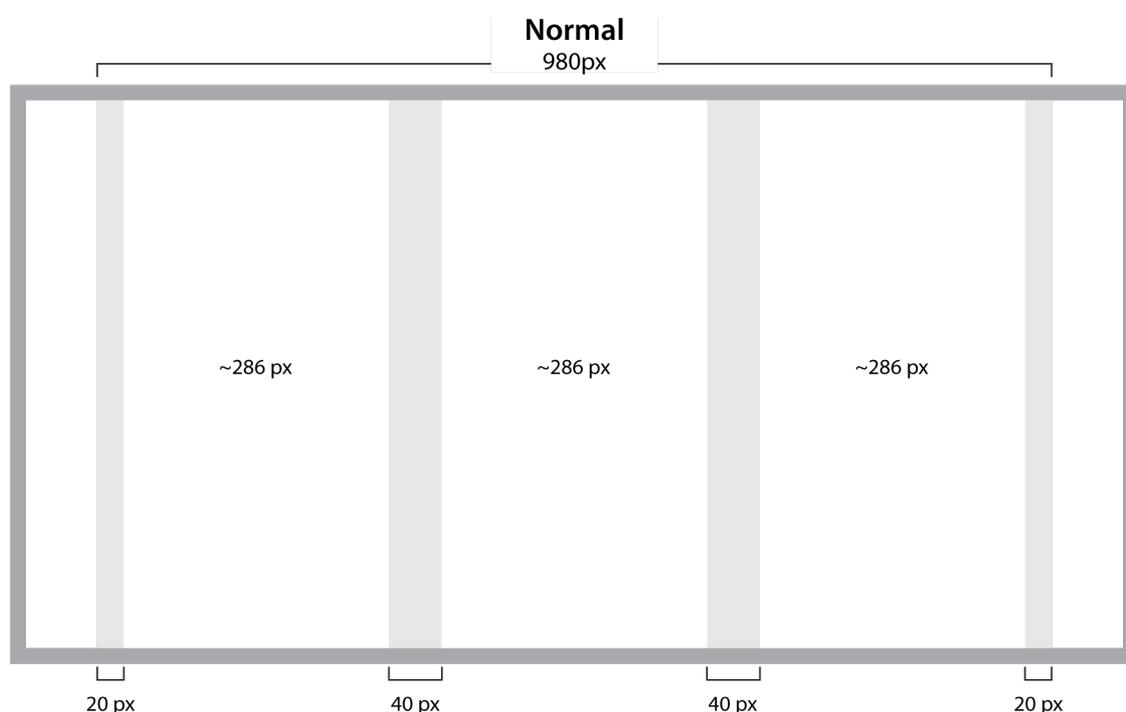


Figure 19: The implemented “frozen” fluid-width grid truly got frozen.

3.5.3 Layout management

As explained in the Regions and Blocks section of the 2.1.1 Fundamental architecture concepts chapter, the built-in Region and Block system in Drupal easily becomes really complex unless the layout structure, and elements within the layout, remain the same across the entire site. The fundamental problem is that regions are global within the theme. It requires a lot of effort and farsightedness to come up with a region system that is maintainable and easy to understand. In addition, blocks by default can only

have one instance, meaning that a piece of content that is placed in the *Header* region, for example, cannot be placed inside *Footer* region at any other part of the site.

The limitations of regions and blocks can be circumvented with the help of contributed modules such as *Context* and *Delta*, or *Panels*. *Context* and *Delta* extend and improve the core region and block system, but still suffer from somewhat limited block configuration options, and unintuitive administration user interface. *Panels* uses its own layout manager that frees from the limitations of the core regions. In addition, *Panels* provides a number of useful features including:

- Fine-grained layout control. *Panels* layouts are reusable plugins that are easy to create. In addition, *Panels* layouts can be nested. This is not possible with core regions.
- Individual pieces of content, *Panel panes*, can be configured *per instance*. For example, controlling the visibility options or applied CSS classes can be done separately to every instance.
- Good Views integration. Using Chaos Tools Suite's Views content panes provides the possibility to pass arguments (contextual filters) to Views displays, and alter parts of the View display configuration (for example number of items to display) from *Panels* pane settings. See Figure 21.
- Ability to override system pages such as node edit page and user profile page.
- Intuitive administration interface (at least compared to the Block Manager administration tool).

The layout management strategy for Costume.fi was to use the region system in conjunction with *Panels* (Figure 20). Only a minimal amount of regions and blocks are used for *common page elements*, while *page specific content* is handed over to *Panels*. Common page elements, such as navigational items, are used throughout the site. The elements can be considered “static” and therefore do not require advanced customization based on their context, for example. Page specific content might vary from page to page, and benefit from *Panels*' system of context and pluggable layouts. For instance, using a different layout based on entity type, or rearranging and configuring elements within the layout.

Figure 20 illustrates the high-level layout structure used in Costume.fi site. The areas under *Region system* are implemented as regular Drupal regions. The *Content* area is

handed over to Panels. Panels enables using different kind of layouts inside the area it is responsible for, as shown in the figure.

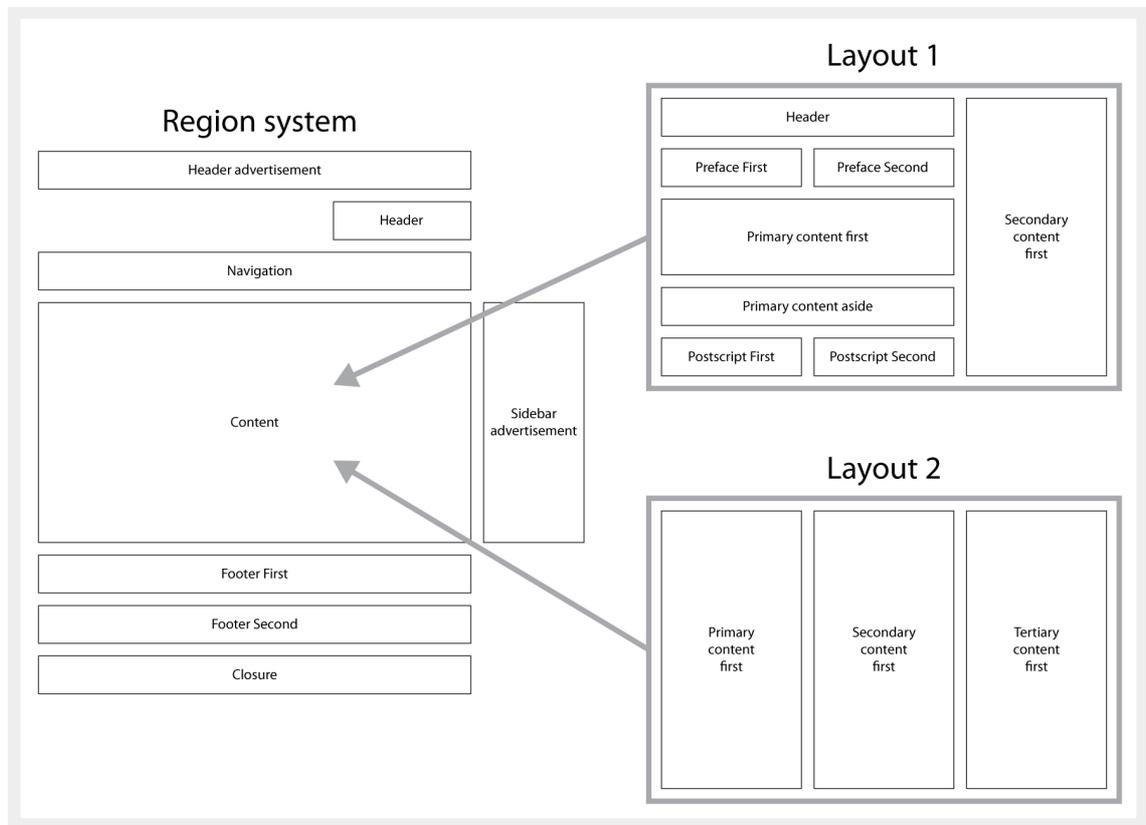


Figure 20: The high-level layout structure is built using region system in conjunction with Panels.

Panels layouts

Panel layouts are plugins used by the Panels module. Panels layouts contain a set of areas, *Panels regions*, where individual pieces of content, *Panels panes*, can be added. Custom Panels layouts can be created using the Panels administrative interface (“flexible” layouts), or in code either inside a theme or a module.

The following will explain how one of the custom Panels layouts used in the *Costume.fi* was implemented. The layout in question, abstractedly named *layout_1*, includes a hefty amount of Panels regions where content such as Views listings, nodes, and forms can be applied using the Panels administrative interface (shown in Figure 21). The figure below shows the *Panel display* configuration for the Fashion talk section front page.

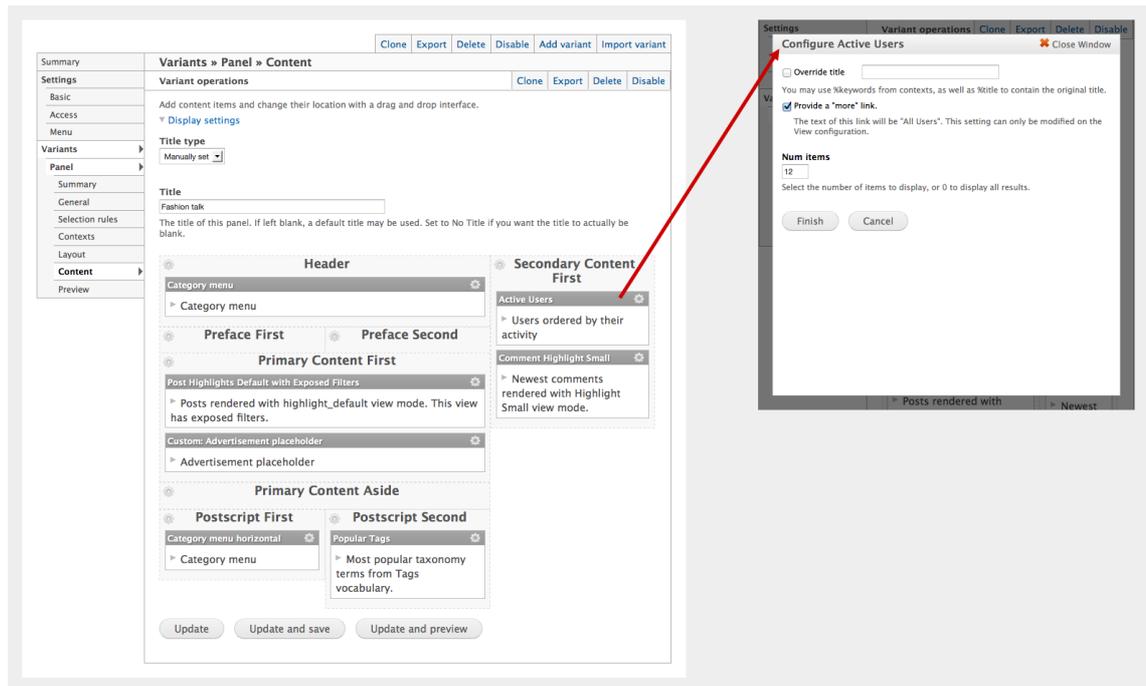


Figure 21: Panels administrative interface: Panels display configuration for Fashion talk section front page. Using Chaos Tools Suite's Views content panes allows altering parts of the View display configuration from Panels pane settings.

The Panels layouts created for Costume.fi were created in the theme. Declaring layouts can be done with a single configuration line (Code example 2). The code below registers a directory inside the theme where custom Panels layouts will be located. Panels then automatically discovers these layouts.

```
1. ; Panels layouts.
2. plugins[panels][layouts] = layouts
```

Code example 2: The location for custom Panels layout can be declared inside a theme with a single configuration line.

The contents of individual Panels layout directory contains:

- layout definition file (e.g. *layout_1.inc*)
- layout template file (e.g. *layout_1.tpl.php*)
- supporting files such as an icon of the layout (for the administrative interface) and CSS files

The layout definition file specifies the properties (e.g. title and category) and regions of the layout. The layout definition of the *layout_1* used in Costume.fi is shown in Code example 3.

```

1. <?php
2. // Plugin definition
3. $plugin = array(
4.   'title' => t('Layout 1'),
5.   'category' => t('Costume'),
6.   'icon' => 'layout_1.png',
7.   'theme' => 'layout_1',
8.   'css' => 'layout_1-admin.css', // admin styles for the layout
9.   'regions' => array(
10.    'header' => t('Header'),
11.    'preface_first' => t('Preface First'),
12.    'preface_second' => t('Preface Second'),
13.    'primary_content_first' => t('Primary Content First'),
14.    'primary_aside_first' => t('Primary Content Aside'),
15.    'secondary_content_first' => t('Secondary Content First'),
16.    'postscript_first' => t('Postscript First'),
17.    'postscript_second' => t('Postscript Second'),
18.   ),
19. );

```

Code example 3: The layout definition file is used to specify properties and regions of a Panels layout file.

The styles enclosed in the defined CSS file (*layout_1-admin.css*), as the name implies, are only used while the page is being assembled in the administrative interface. The actual front-end styles for the layout were written as part of the theme stylesheets, outside of the *layout_1* directory. This was done mainly because it made it easier to rearrange the layout regions within the page layout using media queries, and to use Sass CSS extension to write the stylesheets.

The layout template file is similar to a basic Drupal page template (*page.tpl.php*). The template file specifies how the layout regions are laid out.

```

1. <?php
2. /**
3.  * @file
4.  * Template for Layout 1.
5.  *
6.  * This template provides a two column panel display layout.
7.  */
8. ?>
9. <div class="panel-layout-1 clearfix">
10.   <div class="primary-content">
11.     <?php if ($content['header']): ?>
12.       <header class="panel-panel panel-header clearfix">
13.         <?php print $content['header']; ?>
14.       </header>
15.     <?php endif; ?>
16.
17.     <?php if ($content['preface_first'] || $content['preface_second']): ?>
18.       <section id="prefaces">
19.         <?php if ($content['preface_first']): ?>
20.           <div class="panel-panel panel-preface-first clearfix">
21.             <?php print $content['preface_first']; ?>
22.           </div>

```

```

23.         <?php endif; ?>
24.
25.         <?php if ($content['preface_second']): ?>
26.             <div class="panel-panel panel-preface-second clearfix">
27.                 <?php print $content['preface_second']; ?>
28.             </div>
29.         <?php endif; ?>
30.     </section>
31. <?php endif; ?>
32. </div>
33.
34. <!-- to be continued ... -->
35. </div>

```

Code example 4 shows abbreviated version of the *layout_1.tpl.php* file used in Costume. The full template is available in Appendix 2.

```

36. <?php
37. /**
38.  * @file
39.  * Template for Layout 1.
40.  *
41.  * This template provides a two column panel display layout.
42.  */
43. ?>
44. <div class="panel-layout-1 clearfix">
45.     <div class="primary-content">
46.         <?php if ($content['header']): ?>
47.             <header class="panel-panel panel-header clearfix">
48.                 <?php print $content['header']; ?>
49.             </header>
50.         <?php endif; ?>
51.
52.         <?php if ($content['preface_first'] || $content['preface_second']): ?>
53.             <section id="prefaces">
54.                 <?php if ($content['preface_first']): ?>
55.                     <div class="panel-panel panel-preface-first clearfix">
56.                         <?php print $content['preface_first']; ?>
57.                     </div>
58.                 <?php endif; ?>
59.
60.                 <?php if ($content['preface_second']): ?>
61.                     <div class="panel-panel panel-preface-second clearfix">
62.                         <?php print $content['preface_second']; ?>
63.                     </div>
64.                 <?php endif; ?>
65.             </section>
66.         <?php endif; ?>
67.     </div>
68.
69. <!-- to be continued ... -->
70. </div>

```

Code example 4: Abbreviated version of *layout_1.tpl.php* file.

In conclusion, using Panels for layout management solved a number of issues in the core region and block system. The benefits of using Panels included:

- Ability to choose different layout for different sections or pages.

- Assembling Panels displays is intuitive in the administrative interface.
- Configuring the appearance, functionality, and visibility of Panels panes can be done individually for every instance.
- Creating customized layouts to the exact specifications is easy using Panels layout plugins.

3.5.4 Views listings

Majority of the Costume.fi content listing were created using entity-based output in Views. The entity-based output was used for node (*Post*, *Blog post*, *Competition*, etc.), comment, and user listings.

The approach was selected mainly to circumvent the shortcomings of Views module's output formatting, and for maintainability reasons. Views module was only used as query builder, which in my opinion, is its strong suit.

Rendering entire nodes (as opposed to a set of Views fields) allows modifying the presentation of a single post centrally using node's display configuration screen. Any time a change needs to be made (in the presentation) of a post, it can be made once, and it will happen consistently anywhere the post is used on the site. This weeds out possible inconsistencies in posts effectively. Utilizing the basic Drupal theming concepts (preprocess hooks, templates) ensures majority of the presentational logic stays in the theme level, rather than in database. Mastering the HTML markup is easier using dedicated node templates, and allows using HTML5 elements Views does not support.

The core content of the Costume.fi site is posts produced by the community and editors. Posts are used in different forms and contexts throughout the site. There are a number of variations of post listings such as ones with filtering and sorting options, latest posts, posts by category, and posts by user.

Listings (Figure 22) can mix different types of content. For example, the "Psst...!" (or "Sizzling hot") component in the home page may contain *Post*, *Blog post*, and *Competition* nodes. Individual post may be produced by the editorial staff or the community. Editorial posts are indicated with a red upper right corner. Figure 22 illustrates two examples of posts listings where entity-based output in Views was used.

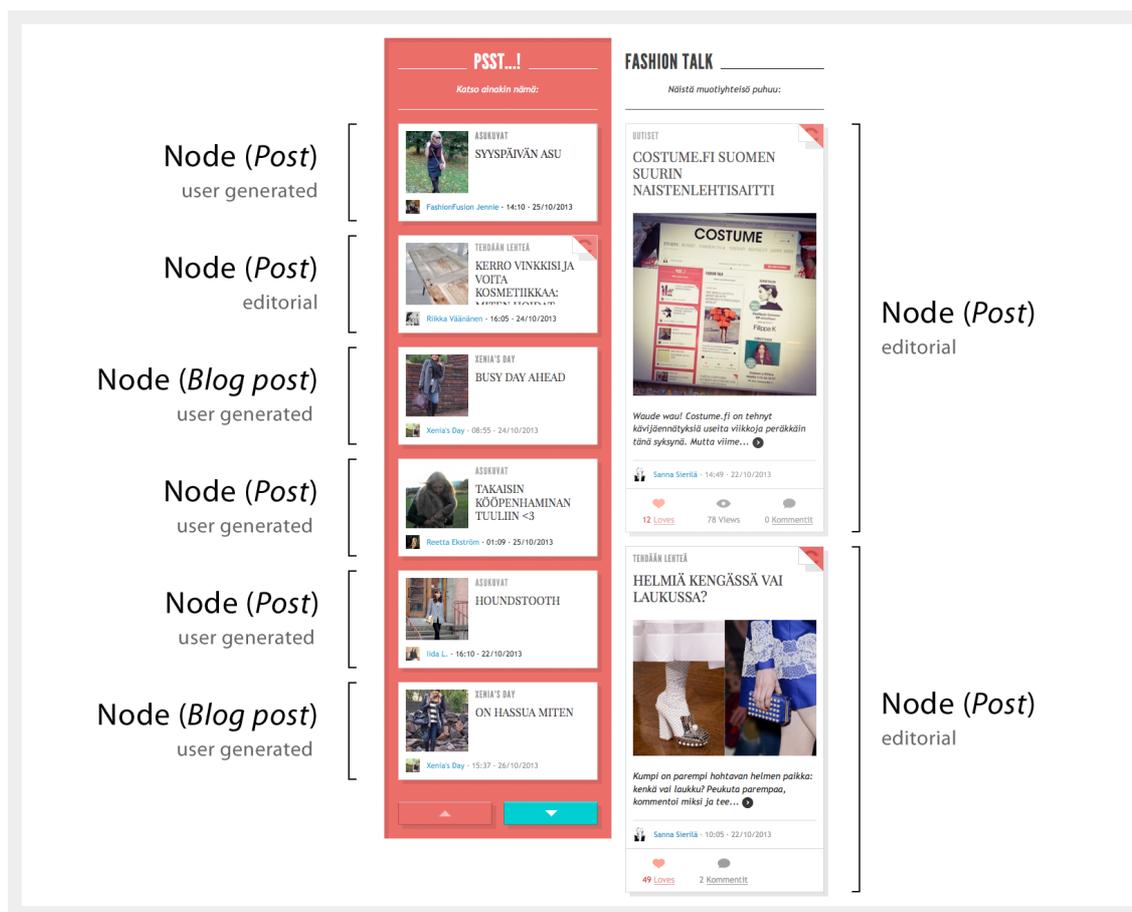


Figure 22: Post listings can contain different kind of content. Individual posts (nodes) inside listings have two distinct representations.

The following sections introduce how individual post nodes were processed for post listings using basic Drupal theming concepts: view modes, theme hook suggestions and preprocessing. The section will cover how to:

- Create custom view modes for nodes
- Configure node display settings per view mode
- Add theme hook suggestions for nodes to create separate node template for each view mode
- Add additional processing for nodes based on their view mode

Custom entity view modes and Post node display settings

Individual post may include pictures, videos, and text, can be tagged, and categorized under a variety of categories. Post content is shown differently depending on the context (Figure 22).

An example of two representations of a post is shown in Figure 23. In the *Highlight Default* representation, majority of the post content is shown to the user: category, title, picture, body text summary, author picture and name, submitted information, and the number of “loves”, views, and comments. *Highlight Small* is an abbreviated presentation where only a subset of available data is shown. Both representations can express special “flags” denoting if the post was authored by a member of the staff; uploaded using mobile application; or if it is a campaign post.

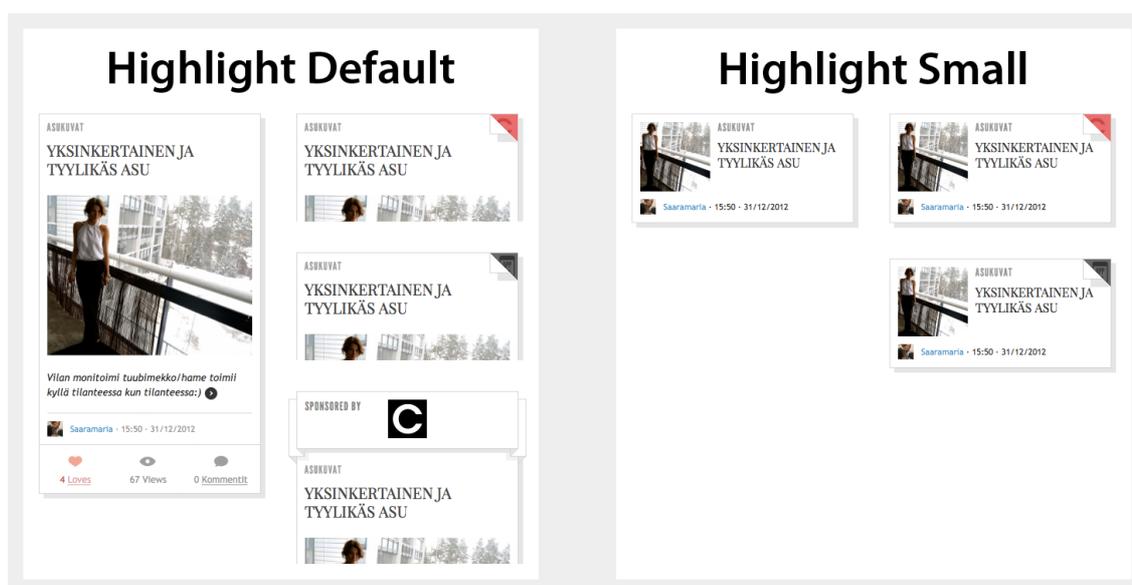


Figure 23: Post content is shown in different ways in different contexts.

These two representations of a post, *Highlight Default* and *Highlight Small*, were implemented as custom entity view modes that were attached to *Post* node type. Entity view modes were defined using a contributed *Entity view modes* module. Managing custom entity view modes is simple using the module’s administration interface (shown in Figure 24). Custom view modes can be created, edited and deleted for a variety of entities. Figure 24 shows the two custom entity view modes created for nodes part of the Costume.fi implementation.

The screenshot displays the administration interface for Entity view modes. It is organized into three main sections: TOKENS, NODE, and FILE.

TOKENS

None (view mode locked)

[Add new view mode](#)

NODE

VIEW MODE	OPERATIONS
Full content	None (view mode locked)
Highlight Default	Edit Delete
Highlight Small	Edit Delete
RSS	None (view mode locked)
Search index	None (view mode locked)
Search result	None (view mode locked)
Teaser	None (view mode locked)
Tokens	None (view mode locked)

[Add new view mode](#)

FILE

VIEW MODE	OPERATIONS
Large	None (view mode locked)

Figure 24: *Entity view modes* module enables managing custom entity view modes using the module's administration interface. Two custom view modes, *Highlight Default* and *Highlight Small*, were created for nodes in *Costume.fi* implementation.

Once the custom entity view modes have been added to the entity registry they will show up in the node *Manage Display* configuration screen (shown in Figure 25). Each view mode may have its own configuration and display order for fields.

The figure displays two screenshots of the Drupal 'Manage Display' interface for 'Post' nodes. The top screenshot is for the 'Highlight Default' view mode, and the bottom screenshot is for the 'Highlight Small' view mode. Both screenshots show a table of fields with their configurations.

FIELD	LABEL	FORMAT	Additional Info
Category	<Hidden>	Plain text	
Main image	<Hidden>	Image	Image style: post_medium Linked to content
Hits	<Hidden>	Radioactivity combo emitter + display	Energy display: Raw numeric value. Emitter is disabled
Hidden			
Body	<Hidden>	<Hidden>	
Images	<Hidden>	<Hidden>	

In the 'Highlight Default' view mode (top), the 'Main image' field uses the 'post_medium' image style, and the 'Hits' field is visible. In the 'Highlight Small' view mode (bottom), the 'Main image' field uses the 'post_thumbnail' image style, and the 'Hits' field is hidden.

Figure 25: Field configuration for *Post* nodes when displayed using *Highlight Default* and *Highlight Small* view modes.

Notable difference in the field configuration between *Highlight Default* and *Highlight Small* view modes is the used image style for *Main image* field. *Highlight Default* view mode uses *post_medium* image style, whereas *post_thumbnail* is used in *Highlight Small* view mode. In addition, the *Hits* field is hidden when the content is being displayed using the latter view mode. Other post node data (shown in Figure 23) are properties of the node, programmatically created “fields” or “flags”, or associated data from other entities, and therefore, cannot be configured using in the *Manage display* administration interface.

Additional Post node properties

One of the programmatically created “flags” denotes if the post was authored by a member of the staff. Drupal offers a set of hooks allowing modules and themes to modify nodes during their build process. A custom `staff_node` flag was added to post nodes as a property using `hook_node_view()` function shown in Code example 5.

```

1. <?php
2. /**
3.  * Implements hook_node_view().
4.  */
5. function costume_util_node_view($node, $view_mode, $langcode) {
6.     $user = user_load($node->uid);
7.
8.     // Adds extra flag to the node if it is authored by a user belonging to
9.     // Costume staff user group.
10.    $node->staff_node = 0;
11.    if (in_array('costume_staff', array_values($user->roles))) {
12.        $node->staff_node = 1;
13.    }
14. }

```

Code example 5: Custom properties may be added to node object using `hook_node_view()`, for instance.

In the function above, the user entity is first loaded using the node author's user id. The user roles of the user are then checked against *costume staff* role. If the user has such role the custom `staff_node` flag flags true. The `staff_node` is stored in the node object being processed. It may be used by other hooks and preprocess functions later in the execution chain, and inside templates.

The `staff_node` flag was used to add a CSS class to editorial nodes' *classes array*. The CSS class was added in the theme layer using `template_preprocess_node()` function.

```

1. /**
2.  * Implements template_preprocess_node(&$variables).
3.  */
4. function costume_preprocess_node(&$variables) {
5.     $node = $variables['node'];
6.
7.     // Adds an extra class to node classes array if editorial node.
8.     if (isset($node->staff_node) && $node->staff_node == 1) {
9.         $variables['classes_array'][] = 'staff-node';
10.    }
11. }

```

Code example 6: `staff_node` flag was used to add a custom CSS class for editorial nodes.

In Code example 6, a custom CSS class "staff-node" is added to nodes' *classes array* if the `staff_node` flag has been set. The CSS class is automatically passed to node template where it is printed as a part of `$classes` variable (see Appendix 1). The class is then used to alter the appearance of the post in stylesheets (red upper right corner).

Post node templates

The output of a node is generated in a template file. Drupal core ships with a generic base node template file (*node.tpl.php*) that defines how nodes are laid out and rendered. The base node template may be overridden in a theme. It is also possible to use alternate, customized templates to theme certain nodes differently than others. (Tomlinson & VanDyk 2010: 200, 207–210.)

Registering an alternate template for discovery is done using Drupal's *theme hook suggestions*. Theme hook suggestions can be thought as naming hints telling the system to pick a template based on certain criteria. Adding theme hook suggestions for nodes is done typically in theme layer using `template_preprocess_node()` function.

As the *Post* nodes had different representations of the content (Highlight Default and Highlight Small view modes, among others), creating separate node template for each view mode seemed appropriate. Template suggestions were added to nodes based on their type and view mode in the theme (*template.php* file) using `template_preprocess_node()` function shown in Code example 7.

```

1. <?php
2. /**
3.  * Implements template_preprocess_node(&$variables).
4.  */
5. function costume_preprocess_node(&$variables) {
6.     // Adds additional theme hook suggestions based on node type and view mode,
7.     // e.g. node--post--highlight_small.tpl.php.
8.     $variables['theme_hook_suggestions'][] =
9.         'node__' . $variables['type'] . '__' . $variables['view_mode'];
10. }

```

Code example 7: Adding theme hook suggestions for nodes can be done in the theme using `template_preprocess_node()` function.

After the additional theme hook suggestions (shown in Code example 7) were added, Drupal was able to use the alternate node templates instead of the base node template. Having separate templates for nodes based on their type and view mode allowed coupling a single representation of a node to a specific template. This, in turn, made it easier to customize the high-level markup, and field and property display order of a node. Without dedicated per-view-mode templates, all the differences would have had to be done using conditional statements inside a common node template for posts, which in turn might have made the template harder to read and maintain. The full node

templates for Post nodes using Highlight Default and Highlight Small view modes are shown in Appendix 1.

Additional processing for nodes based on their view mode

Since the posts were rendered as entire nodes (as opposed to separate Views fields), it allowed additional processing to be done to the node and its content using preprocess and custom functions. As an example, a customized summary of the body text was created programmatically for posts in their Highlight Default view mode. The summary text is constructed in a utility function (Code example 8) that is called during the node build process (Code example 9). The output is then passed to the node template where it is rendered (Appendix 1).

```

1. <?php
2. /**
3.  * Creates a customized summary out of body text field.
4.  *
5.  * @param stdClass $node
6.  *   Node object.
7.  * @param Integer $trim_length
8.  *   Summary text length.
9.  *
10. * @return String
11. *   HTML for body summary field.
12. */
13. function costume_body_summary($node, $trim_length = 110) {
14.     $output = '';
15.
16.     $field_body_items = field_get_items('node', $node, 'body');
17.     if (!empty($field_body_items)) {
18.         $field_body_value = field_view_value(
19.             'node', $node,
20.             'body', $field_body_items[0],
21.             array('type' => 'text_plain')
22.         );
23.
24.         // First strip media tags.
25.         $body_text = costume_strip_media_tags($field_body_value['#markup']);
26.
27.         // Check if there are any actual characters left.
28.         if (strlen(trim(preg_replace('/\xc2\xa0/', ' ', $body_text))) > 0) {
29.             // Trim value to desired length.
30.             $trimmed_field_body =
31.                 truncate_utf8($body_text, $trim_length, TRUE, TRUE);
32.
33.             // Append arrow icon.
34.             $body_summary =
35.                 $trimmed_field_body . '<span class="icon arrow-e"></span>';
36.
37.             // Wrap summary inside a link pointing to node.
38.             $output =
39.                 l($body_summary, 'node/' . $node->nid, array('html' => TRUE));
40.         }
41.     }
42.
43.     return $output;

```

```

44. }
45.
46. /**
47.  * Strips media tags from string.
48.  *
49.  * @param String $text
50.  *   Text to be stripped.
51.  *
52.  * @return String
53.  *   Stripped text.
54.  */
55. function costume_strip_media_tags($text) {
56.     $pattern = "|[[[{\|\/\!]*?[^\\[\]]*?}]|si";
57.     $text = preg_replace($pattern, "", $text);
58.
59.     return $text;
60. }

```

Code example 8: Utility functions developed for creating customized summaries out of node body texts.

In Code example 8, the `costume_body_summary()` function strips unwanted media tags (special markup generated by the *Media* module) from the text and truncates it to desired length. In addition, the summary is wrapped inside a link pointing to the node it belongs to.

```

1. /**
2.  * Implements template_preprocess_node(&$variables).
3.  */
4. function costume_preprocess_node(&$variables) {
5.     $node = $variables['node'];
6.
7.     // Additional processing for Post nodes in Highlight Default view mode.
8.     if ($node->type == 'post'
9.         && $variables['view_mode'] == 'highlight_default') {
10.        // Store customized body summary text for template.
11.        $variables['body_summary'] = costume_body_summary($node, 110);
12.    }
13. }

```

Code example 9: Customized body summary text is passed to the node template in a preprocess function.

In Code example 9, the `costume_body_summary()` function is called if the processed node is of post type in its Highlight Default view mode. The output is stored in `$body_summary` variable that is passed to the node template (see Appendix 1).

In retrospect, all of the processing above for the body text could have been improved by creating a custom *Field formatter* out of it. Using a Field formatter would have generalized the implementation allowing the same processing to be applied to other text fields in other entities as well. Also, it would have removed the need to pass additional

variables to the node template, as the formatter could have been selected for the body text in the Post node *Manage display* configuration screen.

4 Results

4.1 Usage data, user accounts and user-generated content

The statistics in this chapter are gathered from two sources. The usage data is measured using TNS Metrix. TNS Metrix is a service for collecting and analysing site metrics. The metrics are published on a weekly basis at tnsmetrix.tns-gallup.fi. The usage data for Costume.fi has been available in TNS Metrix since around three weeks after the launch.

The user account and user-generated content data have been collected from the core service database. Obvious spam users and content, and data suspected to be corrupt have been excluded from the results.

4.1.1 Usage data

Costume.fi usage data was collected using TNS Metrix. The number of unique weekly visitors within the first 12 months is listed in Table 5 and shown in Figure 26. The number includes all unique visits to the core service and the blogs. Unique visitors is the number of distinct individuals requesting the page during a given period. The metrics provided by TNS Metrix comply with the criteria specified by Audit Bureau of Circulations and the recommendations set by IAB Finland. (Eri kävijät | TNS | SUOMEN WEBSIVUSTOJEN VIIKKOLUVUT.)

The number of unique visitors fluctuate moderately with the average being 62 887, as shown in Table 5 and Figure 26. The lowest amount of visitors was during week 39 of 2012 (53 319), while week 6 of 2013 (75 337) attracted most visitors.

Table 5: Number of unique weekly visitors.

Week	Unique visitors	Week	Unique visitors	Week	Unique visitors
2012 / 37	58487	2013 / 03	71102	2013 / 21	61056
2012 / 38	60560	2013 / 04	68426	2013 / 22	59539
2012 / 39	53319	2013 / 05	63524	2013 / 23	59929

2012 / 40	63583	2013 / 06	75337	2013 / 24	59183
2012 / 41	64444	2013 / 07	62877	2013 / 25	61428
2012 / 42	61482	2013 / 08	64693	2013 / 26	59871
2012 / 43	62969	2013 / 09	67642	2013 / 27	60779
2012 / 44	60443	2013 / 10	63229	2013 / 28	63174
2012 / 45	62414	2013 / 11	61220	2013 / 29	62976
2012 / 46	59912	2013 / 12	62799	2013 / 30	62830
2012 / 47	65694	2013 / 13	60366	2013 / 31	60022
2012 / 48	64225	2013 / 14	62834	2013 / 32	62760
2012 / 49	64497	2013 / 15	62158	2013 / 33	66518
2012 / 50	60030	2013 / 16	60876	2013 / 34	63745
2012 / 51	64188	2013 / 17	61604	2013 / 35	71754
2012 / 52	63214	2013 / 18	62556	2013 / 36	68208
2013 / 01	66008	2013 / 19	57788		
2013 / 02	62499	2013 / 20	59366		

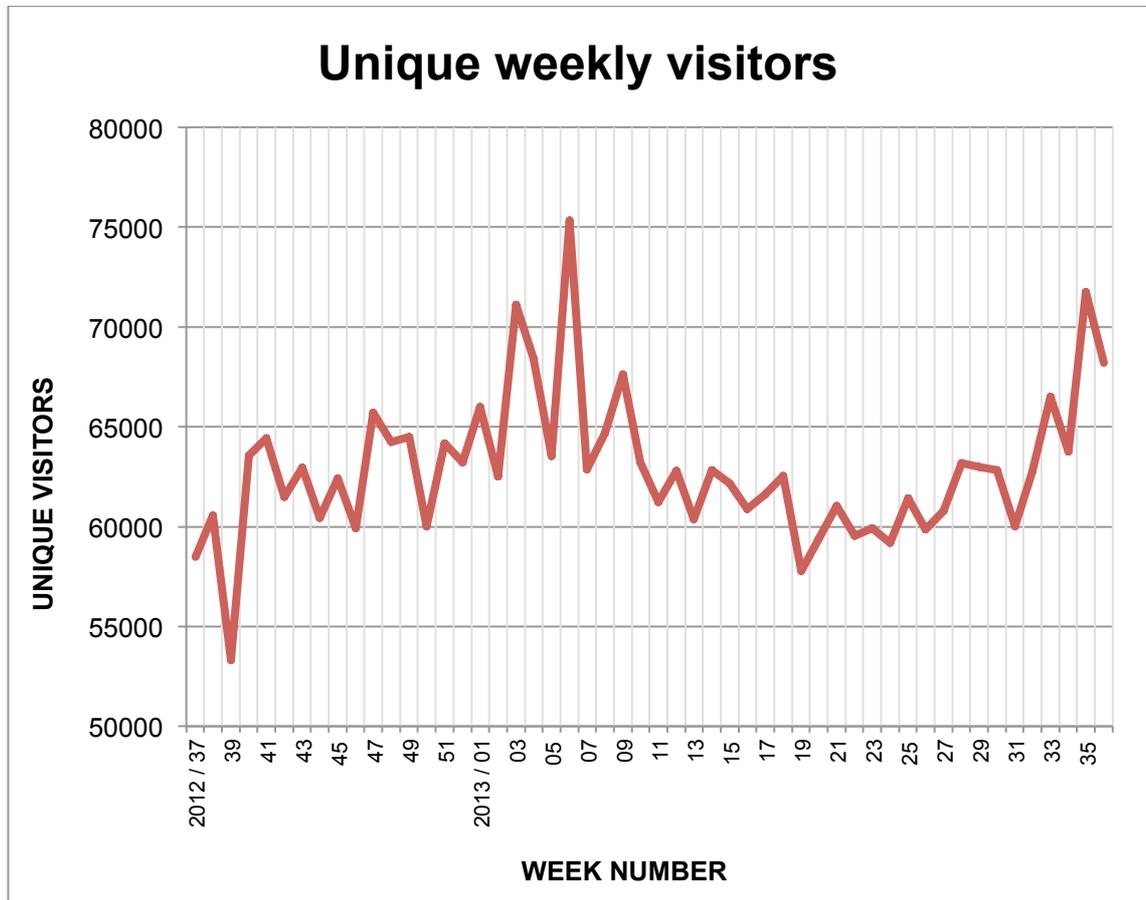


Figure 26: Number of unique weekly visitors.

4.1.2 User accounts

Users are allowed to sign up to the online service by creating a local account, or by connecting their Facebook account with the service. For every new user account, a record is created to the core service database. The number of new user accounts is shown in Figures 27 and 28. User accounts associated with administrative, Costume staff or blogger user roles are not included in the results. In addition, users who have never logged in to the system have been excluded.

The number of new user account registrations per day is shown in Figure 27. According to expectations, majority of the user accounts were registered during the first days after the launch.

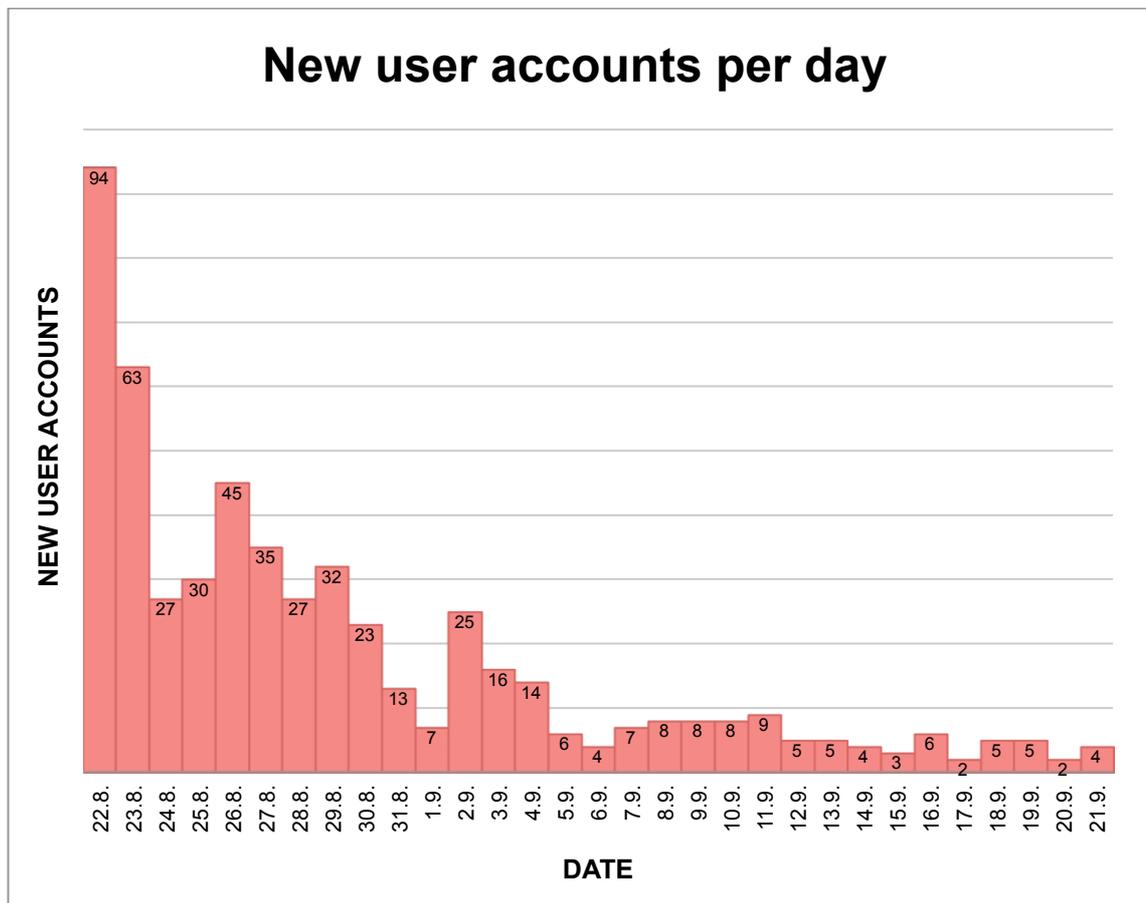


Figure 27: Number of new user account registrations per day within the first 30 days.

The number of new user account registrations during the first 12-month period shows a similar trend, as shown in Figure 28. It is worth noting that the majority of the new user

accounts were created during the first 10 days (between August 22, 2012 and September 1, 2012).

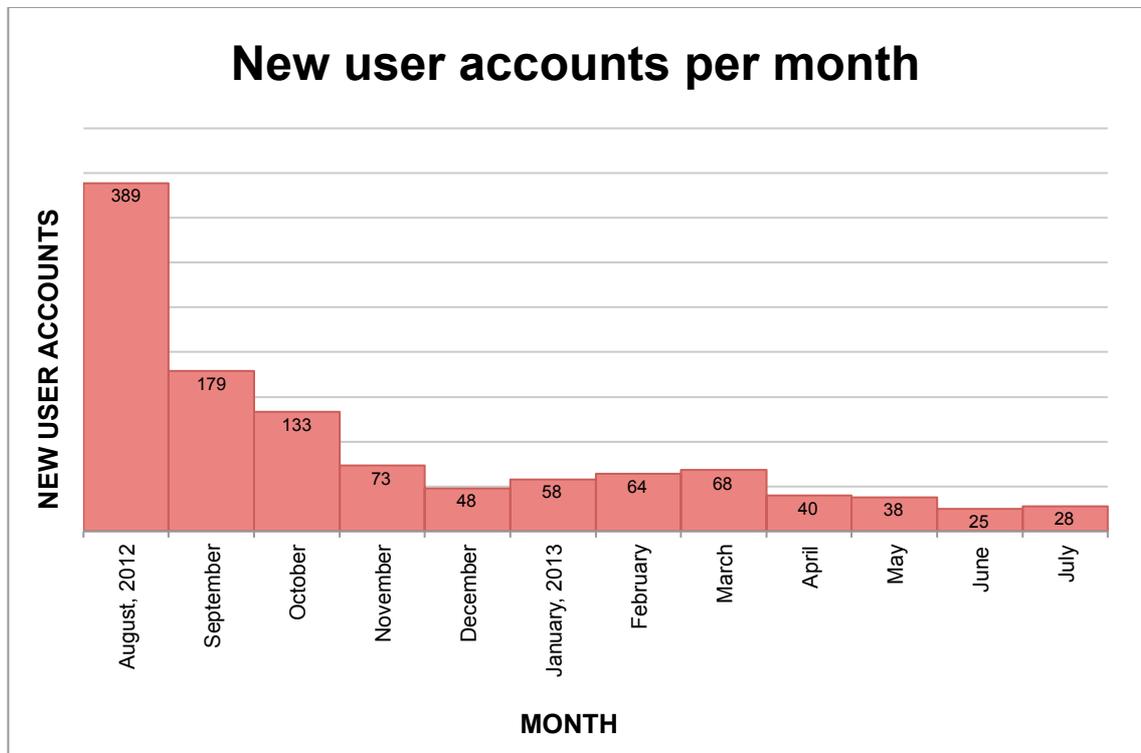


Figure 28: Number of new user account registrations per month within the first 12 months.

4.1.3 User-generated content

The core content of the online service is posts produced by the community and editors. Authenticated users are allowed to publish posts to the Fashion talk section. The number of new posts created by the community is shown in Figure 29. The data only includes nodes of type *Post* that are set as published. Blog entries, and all other types of content have been excluded.

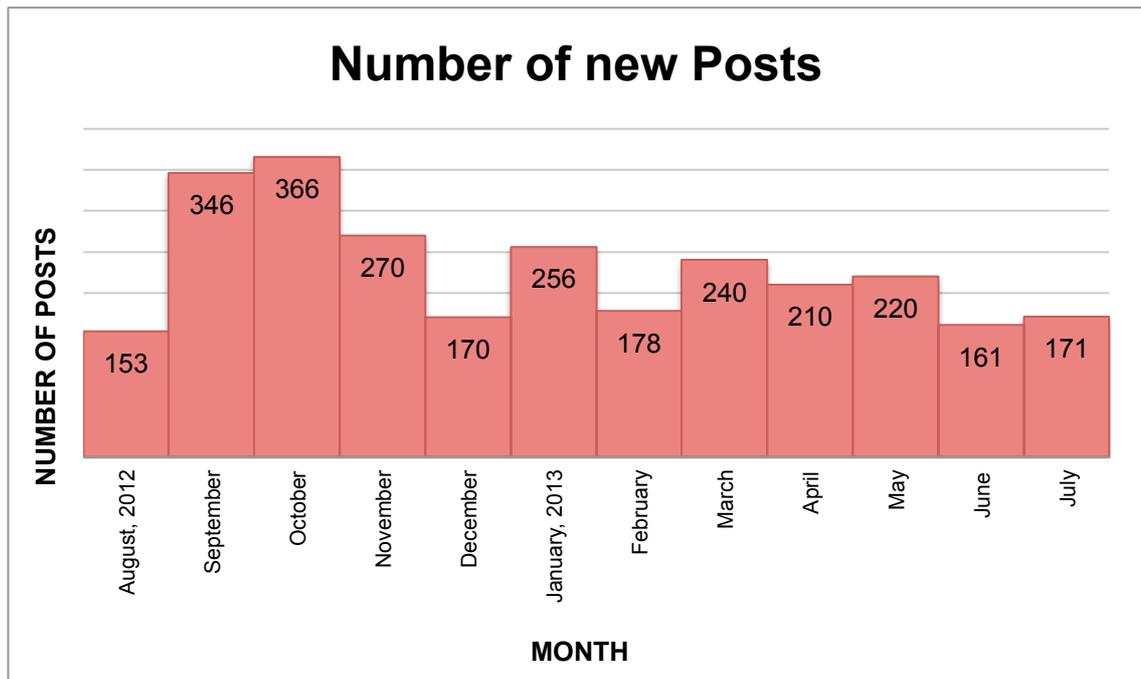


Figure 29: Number of new Posts per month

Individual *Post* node can be categorized under five predefined categories (loose translations in the parenthesis).

- Uutiset (News)
- Kirppis (Buy and sell)
- Asukuvat (Outfit pictures)
- Kysy ja vinkkaa (Questions and tips)
- Tehdään lehteä (Making of the magazine)

The distribution of *Posts* per category is shown in Figure 30. A vast majority of the Posts (around 70 percent) have been categorized under the *Asukuvat* (Outfit photos) category.

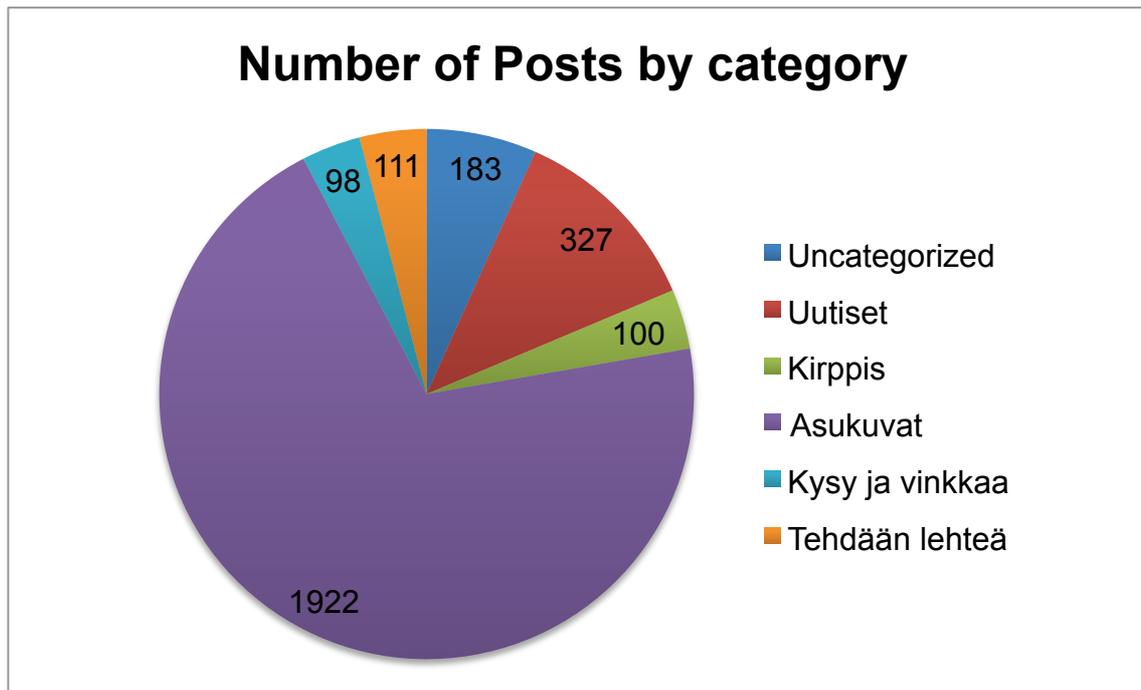


Figure 30: Number of Posts per category during the first 12 months

4.2 Analysis

The online service was released to the public on schedule on August 22, 2012 – the same day the first issue of the paper magazine hit the stores. Helena Jutila, web producer at Bonnier Publications at the time, described the overall result of the project: “The web site has been successful in achieving a look and feel that attracts the desired target group. Easy tools for adding and editing content have inspired the users to actively create content, which has been the key goal of project.” (Costume.fi | Exove.).

The online service was built over the course of the summer of 2012. The original work estimate of ~40 man-days for the technical implementation was exceeded mainly due to changes in scope and some unexpected difficulties with implementation. For example, integrations with third-party systems such as Facebook and advertisement providers took remarkably longer than originally estimated. In addition, the online service was one of Exove’s first fully responsive web sites, and was built partially on top of technology stack (nginx, Fast CGI, and PHP-FPM) that had not been used previously in the company.

The original target of attracting at least 55 000 unique visitors on a weekly basis was met. The number of unique visitors has been fluctuating quite steadily between 60 – 65 000 per week during the first year. The number seems quite impressive, at least when compared to the amount of new accounts registered, and the content created by the community. It is worth remembering that the number of unique visitors retrieved from TNS Metrix include both platforms: the core service and the blogs.

5 Conclusions

This thesis explained the overall process of how Costume.fi was built. The aim of the project was to create the number one community-driven online service in Finland. The online service was launched on schedule, and it met the majority of the original targets in technical terms and operationally.

The selected layout management strategy of complementing the core region and block system using Panels and Chaos Tools Suite's Views Content panes was a success. Having the ability to choose different layout per section, and assembling the pages using reusable components made the site building process future-proof.

Using entity-based Views listing in conjunction with custom view modes allowed keeping the presentational logic in the theme level. The solution addressed many of the issues related to output formatting and maintainability in Views listings, although required more custom coding. Leveraging Drupal's hooks allowed tailoring the presentation of a single entity according to a variety of criteria. In retrospect, some of the custom processing for nodes should have been generalized and implemented as Field Formatters, for example.

Customizing the front-end to the exact specifications was easier using a minimal base theme as a foundation. Utilizing an opinionated, feature-rich base theme instead would have probably ended up in compromises in the high-level layout, and Panels integration.

Prior experience of Drupal, and the support of the development team, was valuable to avoid common pitfalls. Building the theme for Costume.fi required taking a deep dive into how the theme layer, and page loading and rendering process works. As a result, I feel I gained a thorough understanding of the subject.

The user interface was designed to be responsive. Building responsive sites is hard, as it requires taking a wide range of screen resolutions and devices into consideration. On the whole, the front-end build was satisfactory, considering some technologies (e.g. Sass) and techniques were new to me. The custom designed “frozen” fluid-width grid system turned out quite laborious to implement. The needs of the advertisers should have been taken into consideration much sooner, already during the concept design stage.

References

- 1 Adaptive theme. 2009. Web document. Drupal.org. <<https://drupal.org/project/adaptivetheme>>. 15.7.2009. Read on 11.11.2013.
- 2 Boron. 2010. Web document. Drupal.org. <<https://drupal.org/project/boron>>. 24.5.2010. Read on 11.11.2013.
- 3 Buytaert, Dries. 2013. Why the big architectural changes in Drupal 8. Web document. <<http://buytaert.net/why-the-big-architectural-changes-in-drupal-8>>. 9.9.2013. Read on 17.9.2013.
- 4 Drupal 8 Updates and How to Help. 2013. Web document. Drupal.org. <<http://drupal.org/community-initiatives/drupal-core>>. 10.4.2013. Read on 17.4.2013.
- 5 Costume.fi | Exove. Web document. Exove.com. <<http://exove.com/cases/costume-fi>>. Read on 23.6.2014.
- 6 Hodgon, Jennifer. 2013. Programmer's Guide to Drupal. Sebastopol: O'Reilly Media, Inc.
- 7 Korpi, J. 2012. Adaptive Web Design. Metropolia-ammattikorkeakoulu. Bachelor thesis.
- 8 Media Queries. 2012. Web document. W3C. <<http://www.w3.org/TR/css3-mediaqueries/>>. 19.6.2012. Read on 17.4.2013.
- 9 Miles, E., Miles, L. & co. 2011. Drupal's Building Blocks. Craftswordville, Indiana: Pearson Education, Inc.
- 10 Omega. 2009. Web document. Drupal.org. <<https://drupal.org/project/omega>>. 30.6.2009. Read on 11.11.2013.
- 11 Usage Statistics and Market Share of Content Management Systems for Websites, June 2014. 2014. Web document. W3Techs.

- <http://w3techs.com/technologies/overview/content_management/all>. Read on 23.6.2014.
- 12 Shreves, R., Dunwoodie, B. 2011. Drupal 7 Bible. Indianapolis: Wiley Publishing Inc.
 - 13 Starter themes. 2008. Web document. Drupal.org. <<https://www.drupal.org/node/323993>>. 21.10.2008. Read on 24.6.2014.
 - 14 Eri kävijät | TNS | SUOMEN WEBSIVUSTOJEN VIIKKOLUVUT. 2014. Web document. TNS. <<http://tnsmatrix.tns-gallup.fi/public/details/DifferentUsers/2169>>. Read on 9.9.2014.
 - 15 Tomlinson, T., VanDyk, J. 2010. Pro Drupal 7 Development. United States of America: Paul Manning.
 - 16 Website wireframe. 2014. Web document. Wikipedia. <http://en.wikipedia.org/wiki/Website_wireframe>. 28.06.2014. Read on 1.7.2014.
 - 17 Working with Views. 2007. Web document. Drupal.org <<https://www.drupal.org/documentation/modules/views>>. 15.1.2007. Read on 30.6.2014.

Post node templates

```
1. <?php
2. /**
3.  * @file
4.  * Theme implementation to display a Post node in Highlight Default view mode
5.  *
6.  * Extra variables:
7.  * - $author_pictures: User avatar rendered with various image styles.
8.  * - $body_summary: Trimmed body text that is wrapped around a link pointing
9.  *   to node.
10. * - $campaign_node: 1 or 0 depending whether the author of the post belongs
11. *   to campaign_user user group or not.
12. * - $user_full_name: Node author's full name defined in field_full_name found
13. *   in user object.
14. */
15. ?>
16. <?php
17.   hide($content['body']);
18.   hide($content['field_category']);
19.   hide($content['field_hits']);
20.   hide($content['field_tags']);
21.   hide($content['field_uploaded_from_application']);
22.   hide($content['links']);
23. ?>
24. <article id="node-<?php print $node->nid; ?>"
25.   class="<?php print $classes; ?>"<?php print $attributes; ?>>
26.   <?php if (isset($campaign_node) && $campaign_node == 1): ?>
27.     <div class="campaign-banner-wrapper">
28.       <div class="campaign-banner">
29.         <span class="campaign-banner-label">
30.           <?php print t('Sponsored by'); ?>
31.         </span>
32.         <?php if (!empty($author_pictures['small'])): ?>
33.           <div class="campaign-banner-user-picture">
34.             <?php print $author_pictures['medium']; ?>
35.           </div>
36.         <?php endif; ?>
37.       </div>
38.     </div>
39.   <?php endif; ?>
40.
41.   <header>
42.     <?php if (!empty($content['field_category'])): ?>
43.       <?php print render($content['field_category']); ?>
44.     <?php endif; ?>
45.
46.     <?php print render($title_prefix); ?>
47.     <h3<?php print $title_attributes; ?>>
48.       <a href="<?php print $node_url; ?>"><?php print $title; ?></a>
49.     </h3>
50.     <?php print render($title_suffix); ?>
51.   </header>
52.
53.   <div class="content"><?php print $content_attributes; ?>>
54.     <?php print render($content); ?>
55.     <?php if (!empty($body_summary)): ?>
56.       <p><?php print $body_summary; ?></p>
57.     <?php else: ?>
58.       <?php print render($content['body']); ?>
59.     <?php endif; ?>
60.   </div>
```

```
61.
62. <?php if ($display_submitted): ?>
63.   <div class="submitted">
64.     <?php if (!empty($author_pictures['small'])): ?>
65.       <div class="author-picture">
66.         <?php print $author_pictures['small']; ?>
67.       </div>
68.     <?php endif; ?>
69.     <?php if (!empty($user_full_name)): ?>
70.       <?php print $user_full_name; ?>
71.     <?php else: ?>
72.       <?php print $name; ?>
73.     <?php endif; ?> &middot;
74.     <span class="time">
75.       <?php print format_date($created, 'only_time'); ?>
76.     </span> &middot; <span class="date">
77.       <?php print format_date($created, 'only_date'); ?>
78.     </span>
79.   </div>
80. <?php endif; ?>
81.
82. <footer class="sharing">
83.   <?php if (!empty($node->rate_love)): ?>
84.     <div class="rate-loves-wrapper">
85.       <?php print $node->rate_love['#markup']; ?>
86.     </div>
87.   <?php endif; ?>
88.
89.   <?php if (!empty($content['field_hits'])): ?>
90.     <div class="node-views-count-wrapper">
91.       <div><span class="icon node-views"></span></div>
92.       <span class="count node-views">
93.         <?php print render($content['field_hits']); ?>
94.       </span>
95.       <?php print t('Views'); ?>
96.     </div>
97.   <?php endif; ?>
98.
99.   <div class="comment-count-wrapper">
100.     <div><span class="icon speech-bubble"></span></div>
101.     <span class="count comment-count">
102.       <?php print $comment_count; ?>
103.     </span>
104.     <?php print l(t('Comments'), 'node/' . $node->nid, array(
105.       'fragment' => 'comments',
106.     )); ?>
107.   </div>
108. </footer>
109. </article>
```

```
1. <?php
2. /**
3.  * @file
4.  * Theme implementation to display a Post node in Highlight Small view mode.
5.  *
6.  * Extra variables:
7.  * - $author_pictures: User avatar rendered with various image styles.
8.  * - $campaign_node: 1 or 0 depending whether the author of the post belongs
9.  *   to campaign_user user group or not.
10. * - $user_full_name: Node author's full name defined in field_full_name found
11. *   in user object.
12. */
13. ?>
14. <?php
15.   hide($content['field_category']);
16.   hide($content['field_main_image']);
17.   hide($content['field_tags']);
18.   hide($content['field_uploaded_from_application']);
19.   hide($content['links']);
20. ?>
21. <article id="node-<?php print $node->nid; ?>"
22.   class="<?php print $classes; ?>"<?php print $attributes; ?>>
23.   <header>
24.     <?php if (!empty($content['field_main_image'])): ?>
25.       <?php print render($content['field_main_image']); ?>
26.     <?php endif; ?>
27.
28.     <?php if (!empty($content['field_category'])): ?>
29.       <?php print render($content['field_category']); ?>
30.     <?php endif; ?>
31.
32.     <?php print render($title_prefix); ?>
33.     <h3<?php print $title_attributes; ?>>
34.       <a href="<?php print $node_url; ?>"><?php print $title; ?></a>
35.     </h3>
36.     <?php print render($title_suffix); ?>
37.   </header>
38.
39.   <?php if ($display_submitted): ?>
40.     <div class="submitted">
41.       <?php if (!empty($author_pictures['small'])): ?>
42.         <div class="author-picture">
43.           <?php print $author_pictures['small']; ?>
44.         </div>
45.       <?php endif; ?>
46.       <?php if (!empty($user_full_name)): ?>
47.         <?php print $user_full_name; ?>
48.       <?php else: ?>
49.         <?php print $name; ?>
50.       <?php endif; ?> &middot; <span class="time">
51.         <?php print format_date($created, 'only_time'); ?>
52.       </span> &middot; <span class="date">
53.         <?php print format_date($created, 'only_date'); ?>
54.       </span>
55.     </div>
56.   <?php endif; ?>
57. </article>
```

Layout 1 Panels layout template

```
1. <?php
2. /**
3.  * @file
4.  * Template for Layout 1.
5.  *
6.  * This template provides a two column panel display layout.
7.  *
8.  * Variables:
9.  * - $id: An optional CSS id to use for the layout.
10. * - $content: An array of content, each item in the array is keyed to one
11. *   panel of the layout.
12. */
13. ?>
14. <div class="panel-layout-1 clearfix">
15.   <div class="primary-content">
16.     <?php if ($content['header']): ?>
17.       <header class="panel-panel panel-header clearfix">
18.         <?php print $content['header']; ?>
19.       </header>
20.     <?php endif; ?>
21.
22.     <?php if ($content['preface_first'] || $content['preface_second']): ?>
23.       <section id="prefaces">
24.         <?php if ($content['preface_first']): ?>
25.           <div class="panel-panel panel-preface-first clearfix">
26.             <?php print $content['preface_first']; ?>
27.           </div>
28.         <?php endif; ?>
29.
30.         <?php if ($content['preface_second']): ?>
31.           <div class="panel-panel panel-preface-second clearfix">
32.             <?php print $content['preface_second']; ?>
33.           </div>
34.         <?php endif; ?>
35.       </section>
36.     <?php endif; ?>
37.
38.     <?php if ($content['primary_content_first']): ?>
39.       <div class="panel-panel panel-primary-content-first clearfix">
40.         <?php print $content['primary_content_first']; ?>
41.       </div>
42.     <?php endif; ?>
43.
44.     <?php if ($content['primary_aside_first']): ?>
45.       <aside class="panel-panel panel-primary-aside-first clearfix">
46.         <?php print $content['primary_aside_first']; ?>
47.       </aside>
48.     <?php endif; ?>
49.
50.     <?php if ($content['postscript_first'] || $content['postscript_second']):
51.     ?>
52.       <footer>
53.         <?php if ($content['postscript_first']): ?>
54.           <div class="panel-panel panel-postscript-first clearfix">
55.             <?php print $content['postscript_first']; ?>
56.           </div>
57.         <?php endif; ?>
58.
59.         <?php if ($content['postscript_second']): ?>
60.           <div class="panel-panel panel-postscript-second clearfix">
```

```
60.         <?php print $content['postscript_second']; ?>
61.         </div>
62.         <?php endif; ?>
63.     </footer>
64. <?php endif; ?>
65. </div>
66.
67. <?php if ($content['secondary_content_first']): ?>
68.     <div class="secondary-content">
69.         <div class="panel-panel panel-secondary-content-first">
70.             <?php print $content['secondary_content_first']; ?>
71.         </div>
72.     </div>
73. <?php endif; ?>
74. </div>
```