YRKESHÖGSKOLAN
NOVIA

WordPress för utvecklare

Pontus Björkgård

Examensarbete för Tradenom(YH)-examen

Utbildningsprogrammet i Informationsbehandling

Raseborg 2018

EXAMENSARBETE

Författare: Pontus Björkgård

Utbildning och ort: Informationsbehandling, Raseborg

Handledare: Klaus Hansen

Titel: WordPress för utvecklare

Datum 14.11.2018 Sidantal 33 Bilagor 1

Abstrakt

I detta arbete behandlas innehållshanteringssystemet WordPress och centrala koncept inom WordPress utveckling. Det beskrivs också hur systemets olika delar samverkar och hur de webbsidor som besökaren ser produceras.

WordPress använder så kallade teman för att presentera det innehåll som skapas i användargränssnittet. WordPress temasystem tillåter användaren att via användargränssnittet installera och välja vilket tema som ska användas, och på så sätt modifiera hemsidans utseende och funktionalitet. För att utöka WordPress och det aktiva temats funktionalitet används moduler, så kallade plugins. Dessa kan också installeras via användargränssnittet. I arbetet förklaras hur en del av de APIs som WordPress erbjuder kan användas för att utöka systemet via plugins och teman och API funktionernas användning behandlas grundligt. WordPress framtid, Gutenberg, presenteras också kort.

Språk: Svenska Nyckelord: WordPress, tema, plugin, webbutveckling

---

BACHELOR'S THESIS

Author: Pontus Björkgård
Degree Programme: Business Information Technology, Raasepori
Supervisor(s): Klaus Hansen

Title: WordPress for Developers

Date 14.11.2018 Number of pages 33 Appendices 1

Abstract

This thesis covers the WordPress content management system and central concepts within WordPress development. I also describe how different parts of the system interact with each other and how the webpages that are sent to the user are produced.

WordPress uses so called themes for presenting the content made in the user interface. The WordPress theme system allows users to install and choose what theme to use, and in this way modify the look and functionality of the website. Modules, or so-called plugins, are used to extend the functionality of WordPress and the active theme. These can also be installed via the user interface. This thesis covers how some of the APIs that WordPress offers can be used to extend the system via plugins and themes. The functions of these APIs and their uses are thoroughly covered. The future of WordPress, Gutenberg, is also briefly covered.

Language: Swedish Key words: WordPress, theme, plugin, webdevelopment

---

Innehållsförteckning

Bilageförteckning

1 Inledning

Moderna innehållshanteringssystem gör processen att skapa och underhålla hemsidor snabb och enkel. Att skriva en hemsida med statisk HTML är tidskrävande och svår för personer utan kodningskunskap att uppdatera. Ett innehållshanteringssystem använder sig av ett programmeringsspråk och en databas för att skapa dynamiska sidor på frontenden. Detta innebär att sidornas utseende och layout är separerade från sidornas innehåll, så att utseendet kontrolleras av mallar och innehållet tas från databasen. Innehållet på en hemsida byggd med ett innehållshanteringssystem redigeras via ett användargränssnitt, vilket för de flesta är mycket smidigare än att redigera ett HTML dokument (Vining 2008).

Dessa innehållshanteringssystem erbjuder främst möjligheter för enkel publicering av innehåll, men också ofta användbara APIs som tillåter utvecklare att modifiera och utöka systemens funktionalitet.

1.1 Syfte

I detta arbete berättar jag om det populära innehållshanteringssystemet WordPress och utforskar vilka möjligheter som erbjuds utvecklare.

Syftet med detta arbete är utveckla mina kunskaper i WordPress. Jag har tidigare jobbat en del inom WordPress, främst med användargränssnittet, men jag känner att det finns en hel del hål i min kunskap vad gäller systemet från en utvecklares synvinkel. Jag planerar att ta reda på hur de APIs som WordPress erbjuder kan användas för att utöka innehållshanteringssystemets funktionalitet.

För att hjälpa mig uppnå syftet har jag installerat WordPress på en lokal XAMPP server. Detta eftersom det är smidigare att jobba med filer som finns lokalt än över FTP. Jag har skapat ett tema (teman förklaras i kapitel 3.2), i vilket jag testar mig fram med olika APIs för att utöka och modifiera systemets funktionalitet. Jag använder mig av textredigeraren Atom, eftersom denna redigerare erbjuder en rejäl mängd paket som underlättar WordPress-utveckling, såsom auto-complete av funktioner.

2 WordPress

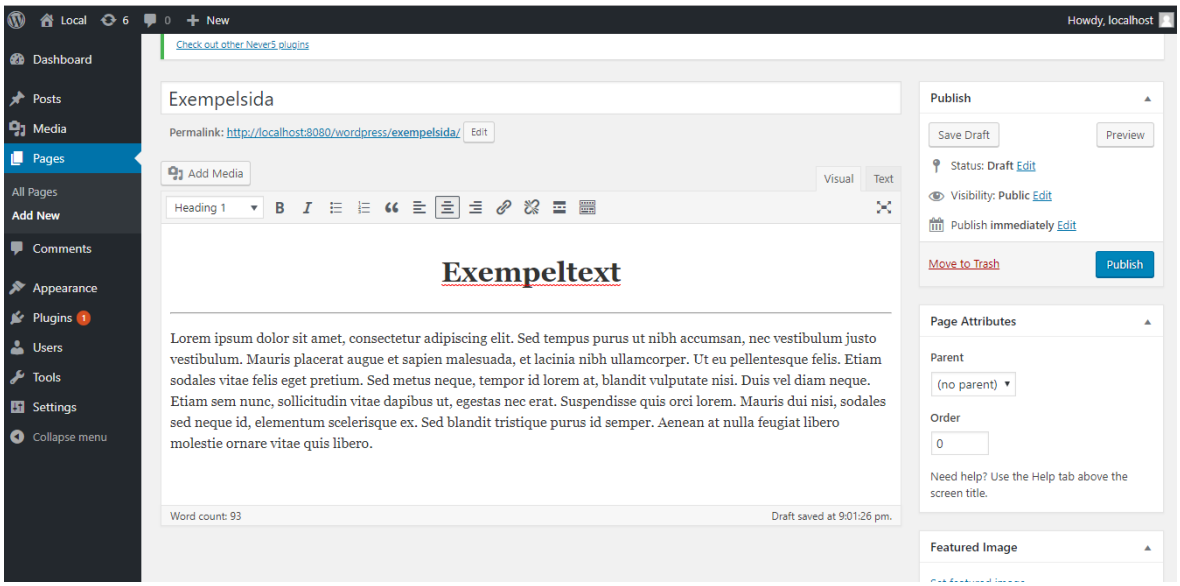
WordPress är ett innehållshanteringssystem ämnad för utveckling, publicering och uppehåll av hemsidor och bloggar. WordPress används av ca. 30 % av alla webbplatser på internet och är världens populäraste innehållshanteringssystem. WordPress är skrivet i PHP och använder databashanteringssystemet MySQL för lagring av information. Dess källkod är öppen, vilket innebär att den är tillgänglig att använda, läsa, modifiera och vidaredistribuera för vem som helst (iThemes u.å.).

Utvecklingen av WordPress påbörjades år 2003 av Mike Little och Matt Mullenweg som ville skapa en ny plattform baserad på bloggplattformen b2/cafelog. I maj 2003 släpptes version 0.7 som en direkt uppföljare till b2/cafelog version 0.6 (WhoIsHostingThis u.å.).

2.1 Användargränssnittet

WordPress erbjuder ett användargränssnitt i vilket användaren kan hantera olika aspekter av hemsidan, såsom innehåll och design (iThemes u.å.).

I Figur 1 ser man hur användargränssnittet ser ut när man skapar en ny sida. Till vänster ser man alla tillgängliga menyflikar under vilka finns sidor på vilka man kan redigera olika aspekter av hemsidans innehåll och design.



Figur 1. WordPress användargränssnitt.

2.1.1 Dashboard

När man först loggar in i användargränssnittet möts man av Home-sidan som finns under Dashboard-fliken. Denna sida fungerar som en översikt och ger diverse information om hemsidan, exempelvis antalet publicerade sidor och inlägg. Här finns också genvägar till de mest väsentliga delarna av användargränssnittet. På Updates-sidan hittas information om eventuella uppdateringar.

2.1.2 Posts

På Posts-sidan har man möjlighet att skapa, radera, redigera och publicera innehåll av inläggstypen Post.

I WordPress innebär Post types, eller inläggstyper, olika typer av innehåll. WordPress är ursprungligen en bloggplattform, som med tiden utvecklades till ett innehållshanteringssystem som erbjuder skapandet av andra typer av innehåll än blogginlägg, andra inläggstyper. WordPress erbjuder inläggstyperna Post, Page, Attachment, Revision och Nav Menu, men möjligheten att skapa egna inläggstyper, så kallade Custom post types, eller anpassade inläggstyper, finns (WPBeginner (WPB) 2016).

Redigeringsgränssnittet fungerar på samma sätt för inläggstypen Page, och för anpassade inläggstyper. Sidans texter redigeras via en textredigerare som heter TinyMCE (WordPress Codex (WPC) u.å.a).

2.1.3 Media

Under Media-fliken finns två sidor, en för uppladdning av filer via ett Drag & drop gränssnitt, och en för hantering och redigering av dessa filer.

I samband med redigering eller publicering av ett inlägg finns vanligtvis möjlighet att ange en utvald bild. Denna bild kan via redigeringsgränssnittet laddas upp eller väljas från mediabiblioteket, och den media som laddas upp i samband med publiceringen/ redigeringen finns sedan tillgänglig i biblioteket.

2.1.4 Comments

På Comments-sidan hanteras kommentarer som besökare har skrivit. På denna sida har administratören möjlighet att bl.a. godkänna, eller inte godkänna, kommentarer.

Ett inläggs kommentarer kan aktiveras eller inaktiveras i inläggets redigeringsvy, och under Settings-fliken finns en sida med inställningar för bl.a. global aktivering/ inaktivering av kommentarer och diverse inställningar för moderation.

2.1.5 Appearance

Under Appearance hanteras teman, widgets, navigationsmenyer, och via ett gränssnitt som heter Customizer, diverse temainställningar.

Ett tema är en samling av filer som avgör funktionalitet och hur innehållet som skapas i WordPress presenteras på frontenden (WordPress Developer Resources (WPDR) u.å.o). Det finns tusentals gratis teman i WordPress temakatalog, och från tjänster såsom Theme Forest kan man köpa mera avancerade teman.

Widgets gör det möjligt att lägga till innehåll och funktionalitet till hemsidans så kallade widget-ready areas, eller widgetområden. I de flesta teman finns dessa områden i sidofält och i sidfötter. WordPress erbjuder ett antal widgets för olika ändamål, och via teman och plugins kan listan på tillgängliga widgets utökas (WPB u.å.b).

Navigationsmenyer är ”inlägg” av inläggstypen Nav Menu, och består av antingen länkar till sidor på hemsidan eller skräddarsydda länkar.

Under Appearance-fliken finns också en kodredigerare i vilken man kan redigera temafiler.

2.1.6 Plugins

WordPress plugins är tilläggsmjukvara som utökar den funktionalitet som WordPress och det aktiva temat erbjuder. I plugin katalogen, och från diverse andra tjänster, hittar man tusentals plugins som erbjuder möjligheter för allt från skapandet av bildgallerier till skapandet och hantering av webbutiker (WPB 2017).

Under Plugins-fliken har man möjlighet att hantera installerade plugins. Man har också tillgång till WordPress pluginkatalog, och ett inbyggt verktyg gör installationsprocessen

snabb och enkel. Det är också möjligt att installera plugins genom att manuellt ladda upp filerna.

2.1.7 Users

WordPress erbjuder hemsidans ägare möjligheten att skapa nya användarkonton, och hantera dessa användares privilegier. Detta sköts under Users-fliken.

När en användare skapas tilldelas denne en användarroll. WordPress erbjuder fem olika roller med varierande privilegier. Användare med rollen administratör är de ända som har hantera användare.

Användare har möjlighet att ange bl.a. smeknamn, profilbild och en beskrivning, som kan visas på frontenden, exempelvis i samband med inlägg denne har skrivit.

2.1.8 Tools

På Tools-sidan finns användbara verktyg för att exempelvis exportera data, eller importera data från andra innehållshanteringssystem.

2.1.9 Settings

Under Settings-fliken sköts hemsidans inställningar. WordPress erbjuder en stor mängd inställningar för bl.a. hemsidans titel och beskrivning, tidsformat, inställningar angående uppladdning och hantering av media, inläggsformatering, permalänkstruktur och alternativ för Post via email funktionalitet.

3 Vad består WordPress av?

WordPress kan grovt delas in i tre komponenter: core (kärnan), teman och plugins. Dessa komponenter är separerade från varandra (Pataki 2017).

En WordPress installations rotmapp innehåller tre huvudmappar: wp-admin, wp-content och wp-includes. I rotmappen finns förutom dessa mappar ett antal filer för exempelvis konfigurering av databasinställningar och permalänkstruktur (Ewer 2016).

3.1 Core (kärnan)

Kärnan innebär de filer som WordPress i grunden består av, och erbjuder den funktionalitet som utgör WordPress, såsom användargränssnittet. WordPress kärnan erbjuder möjligheten för användaren att skapa innehåll och ladda upp media via ett gränssnitt, men det är upp till temat och plugins att presentera innehållet på frontenden (Pataki 2017). De filer som hör till WordPress kärnan hittas i installationens wp-admin och wp-includes mappar.

3.2 Teman

Ett WordPress tema innehåller vanligtvis tre olika typer av filer:

1. Mallfiler
2. Stylesheet (style.css)
3. Functions fil (functions.php)

Ett temas filer förvaras inuti en mapp inuti wp-content/themes.

3.2.1 Mallfiler

Mallfiler är återanvändbara, modulära filer som WordPress använder för att skapa de dynamiska webbsidor som besöks på frontenden (WPDR u.å.j).. I mallfilerna används PHP funktioner, så kallade template tags för att hämta innehåll från databasen (WPDR u.å.l).

Att sidorna är dynamiska innebär att WordPress använder sig av information som fås från den begärda URLens querysträng för att avgöra vilket innehåll som ska hämtas från databasen. WordPress avgör också vilka mallfiler som ska användas baserat på information som fås från querysträngen. Ett tema innehåller oftast ett antal olika mallfiler som används för att skriva ut olika sorters sidor, och när en URL begärs söker WordPress efter en passande mallfil för det begärda innehållet enligt Template hierarchy, eller mallhierarkin (WPDR u.å.k).

En permalänken för ett exempelinlägg vid namn Exempel 1 av inläggstypen Exempeltyp ser ut som följande:

http://exempel.com/?exempeltyp=exempel-1

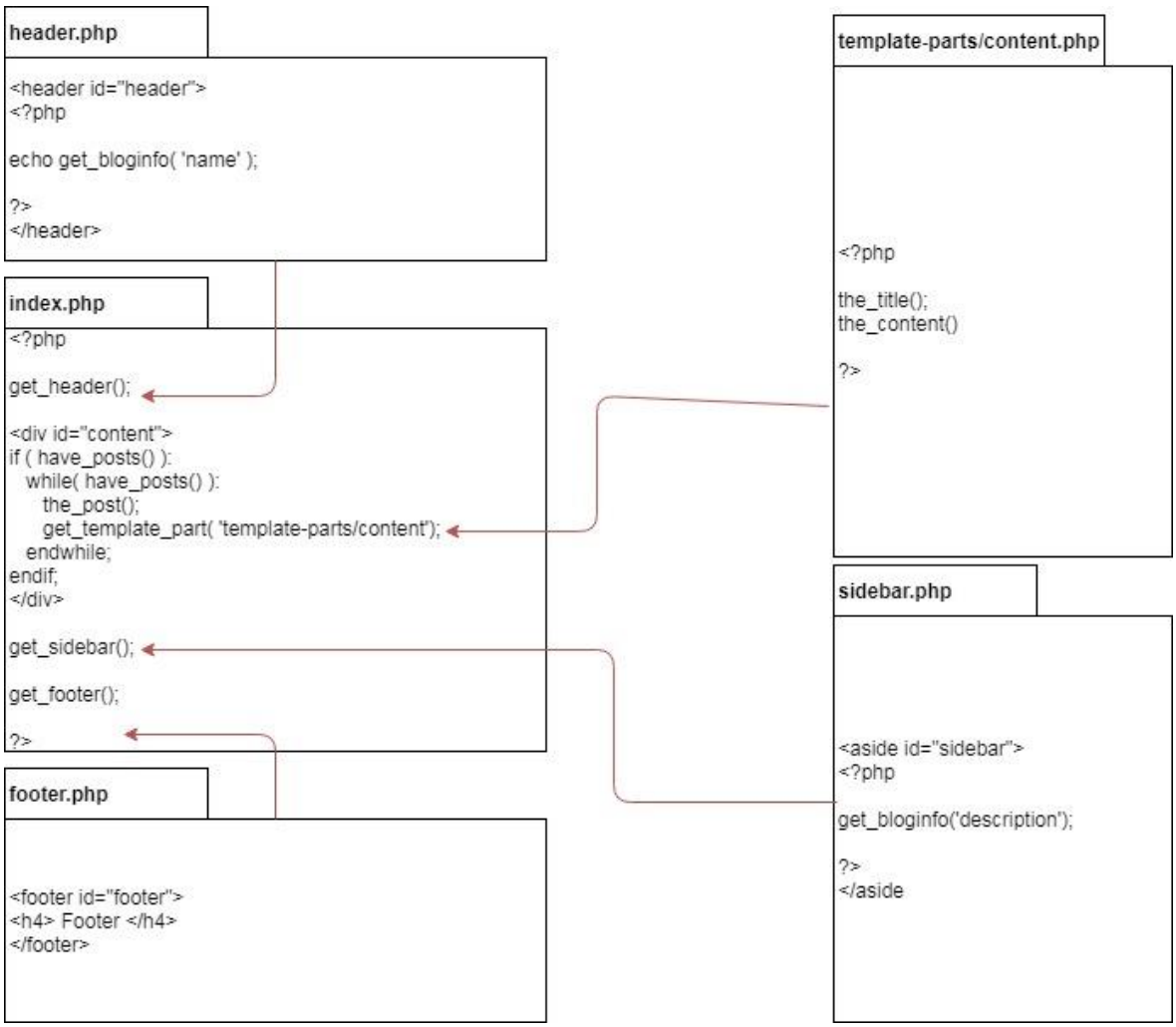
Till querysträngen hör texten efter frågetecknet. När denna URL begärs hämtar WordPress innehåll från databasen som är kopplat till inläggets slug, vilken i detta fall är ”exempel-1”. En slug är en URL giltig version av ett namn (WPB u.å.a). I bilaga 1 beskrivs hur WordPress använder mallhierarkin för att hitta en passande mallfil för inlägget Exempel 1.

WordPress erbjuder möjligheten att använda Apache modulen mod_rewrite för att modifiera permalänkar. På Permalinks-sidan under Settings-fliken i användargränssnittet finns dessa inställningar, som skapar mod_rewrite regler i installationens .htaccess fil (WPC u.å.h). När en URL begärs modifieras den enligt dessa regler, till att exempelvis se ut som följande:

http://exempel.com/exempeltyp/exempel-1

Vanligtvis används vissa element på en hemsida oberoende av vilken sida besökaren befinner sig på. Koden som skriver ut dessa element, exempelvis sidhuvuden och sidfötter, skrivs vanligtvis ner i så kallade template partials filer. Template partials används för att förhindra upprepning av kod (WPDR u.å.j).

I figur 2 ser man hur ett temas mallfiler kan pusslas ihop för att skriva ut en webbsida. En URL har begärts, och WordPress har konstaterat att index.php är den mest passande mallfilen som hittas bland temafilerna. Detta exempeltemas simpla index.php fil innehåller ett antal template tags som hämtar innehåll och skriver ut sidan. `get_header()`, `get_footer()`, `get_template_part()` och `get_sidebar()` är template tags som används för att inkludera filer. Alla dessa funktioner gör i princip samma sak, men ändå används de olika. WordPress erbjuder skilda template tags för inkludering av de tre vanligaste delarna av en hemsida eller blogg, nämligen sidhuvud, sidofält och sidfot. Funktionen `get_header()` inkluderar filen header.php, `get_footer()` filen footer.php, och `get_sidebar()` filen sidebar.php. Ifall dessa filer inte hittas i temat hämtas de från kärnan. Funktionen `get_template_part()` gör i princip samma sak, men är avsedd till att inkludera skräddarsydda template partials. Funktionerna `get_header()` och `get_template_part('header')` hämtar alltså samma fil. `get_blog_info()` används för att hämta diverse information om hemsidan (WPDR u.å.j).

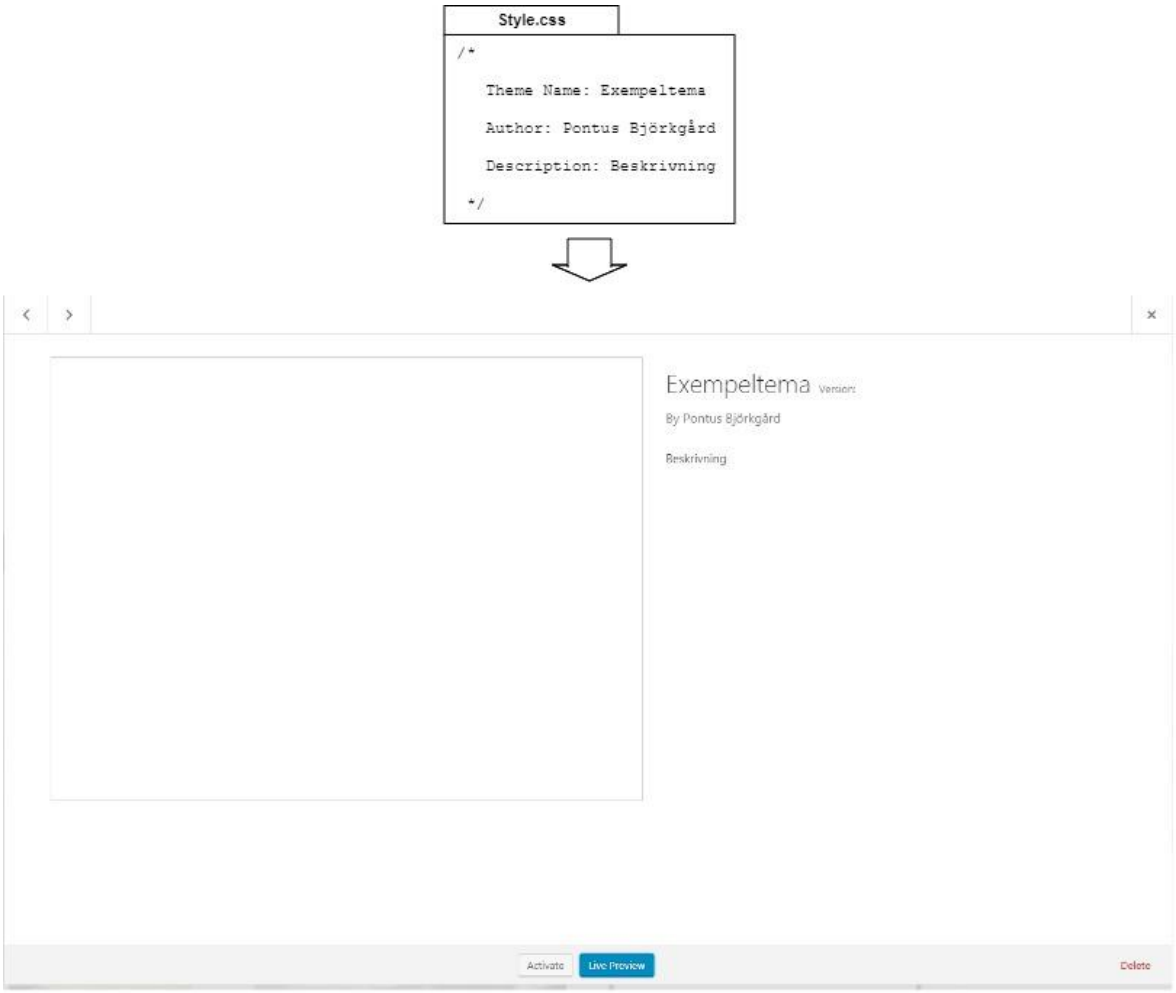


Figur 2. Användning av mallfiler

För att visa inlägg använder sig WordPress av en while loop, som kallas The Loop. I denna loop bearbetas alla de inlägg som begärts. Detta gör det möjligt för WordPress att använda samma mallfiler för enskilda inlägg som för arkiv innehållande flera inlägg. I detta exempeltemas index.php kontrolleras först ifall inlägget/inläggen som begärs existerar. Ifall `have_posts()` returnerar `true` påbörjas loopen, som kör en iteration per inlägg. Funktionen `the_post()` gör det möjligt att med template tags skriva ut information om det aktuella inlägget i loopen (Muldoon 2014). I detta exempel finns template tag funktionerna `the_title()` och `the_content()` i filen content.php, som hämtas med funktionen `get_template_part()`. Dessa template tags skriver ut titeln och innehållet för iterationens inlägg.

3.2.2 Style.css

Ett tema kräver filen style.css för att fungera. Style.css behöver inte nödvändigtvis innehålla CSS, utan endast en kommentar högst uppe i filen. WordPress behöver denna kommentar för att tolka samlingen av filer som ett tema och därmed göra temat aktiverbart. WordPress använder informationen i kommentaren för att presentera temats namn, beskrivning mm. i temasektionen i användargränssnittet (Se figur 3) (WPDR u.å.f).



Figur 3. Hur informationen i ett temas style.css fil presenteras i användargränssnittet

3.2.3 Functions.php

I ett temas functions.php fil definieras temats funktionalitet. WordPress kärnan innehåller funktioner för exempelvis skapandet av anpassningsbara menyer och sidofält, och aktivering av möjligheten att ange en utvald bild för inlägg. Man kan även skriva sina egna funktioner och med hjälp av ett antal olika APIs som WordPress erbjuder utöka användargränssnittet och anpassa hur innehållet presenteras på frontenden (WPC u.å.c).

Två funktioner som ofta hittas i denna fil är `add_theme_support()` och `wp_enqueue_style()`. Funktionen `add_theme_support()` används för att aktivera stöd för viss funktionalitet som kärnar erbjuder (WPDR u.å.b). Att använda funktionen `wp_enqueue_style()` är det rekommenderade sättet att ladda CSS filer i webbsidornas `<head>` sektion (WPDR u.å.p).

3.2.4 Child teman

Det rekommenderas inte att direkt redigera temafiler, ifall man inte själv är temats utvecklare. Detta är p.g.a. att ändringar man gjort går förlorade ifall temat uppdateras. Lösningen på detta problem är Child teman, eller barnteman. Ett barntema är ett tema som ärver filer från ett annat tema, ett Parent tema (föräldertema) (WPC u.å.b).

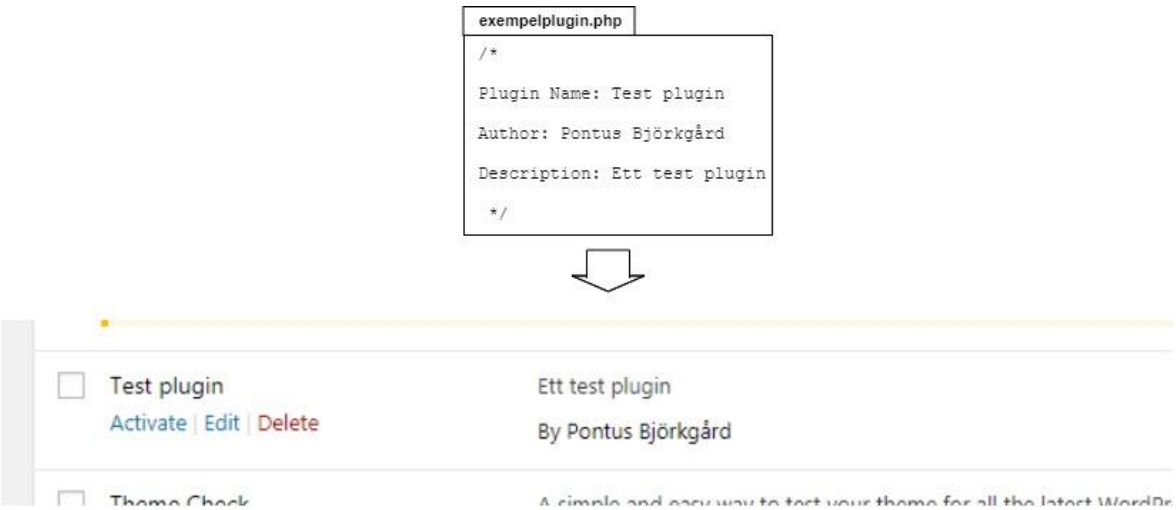
Ett barntema bör innehålla åtminstone två filer: style.css och functions.php, som placeras i en mapp inuti wp-content/themes mappen. Såsom i varje tema bör style.css filen innehålla kommentarer med information. För att temat skall fungera som ett barntema krävs det att kommentaren innehåller en rad med följande information: ”Template: föräldertema”, där ”föräldertema” är namnet på mappen i vilken föräldertemat finns (WPC u.å.b).

I functions.php filen bör föräldertemats style.css, och övriga CSS filer som temat använder, laddas med funktionen `wp_enqueue_style()`. Ett barntema använder som standard sitt föräldertemas mallfiler, men överskrivning av en mallfil är möjligt genom att i barntemat skapa en fil med samma namn (WPC u.å.b).

3.3 Plugins

WordPress plugins är moduler som möjliggör ändringar och utökningar av den funktionalitet som WordPress kärnan och det aktiverade temat erbjuder. Ett WordPress plugin bör innehålla åtminstone en PHP fil, och på samma sätt som i ett temas style.css fil bör information anges högst uppe i filen i form av en kommentar. Denna kommentar möjliggör pluginaktivering i användargränssnittet (Pataki 2011).

Pluginfiler förvaras i en mapp med samma namn som pluginets huvudfilen inuti wp-content/plugins. I figur 4 ser man hur informationen i ett plugins huvudfil presenteras i användargränssnittet.



Figur 4. Hur informationen i ett plugins huvudfil presenteras i användargränssnittet.

Teman och plugins är separerade från varandra. Detta gör användning av ett plugin möjligt med vilket tema som helst. Funktionaliteten som ett plugin erbjuder förblir oförändrad fastän användaren aktiverar ett nytt tema.

4 WordPress API

WordPress API innehåller ett antal API sektioner som utvecklare kan använda för att utöka WordPress kärnans funktionalitet (WPC u.å.j). I detta kapitel tänker jag gå djupare in på några av de olika APIs som WordPress erbjuder.

4.1 Plugin API

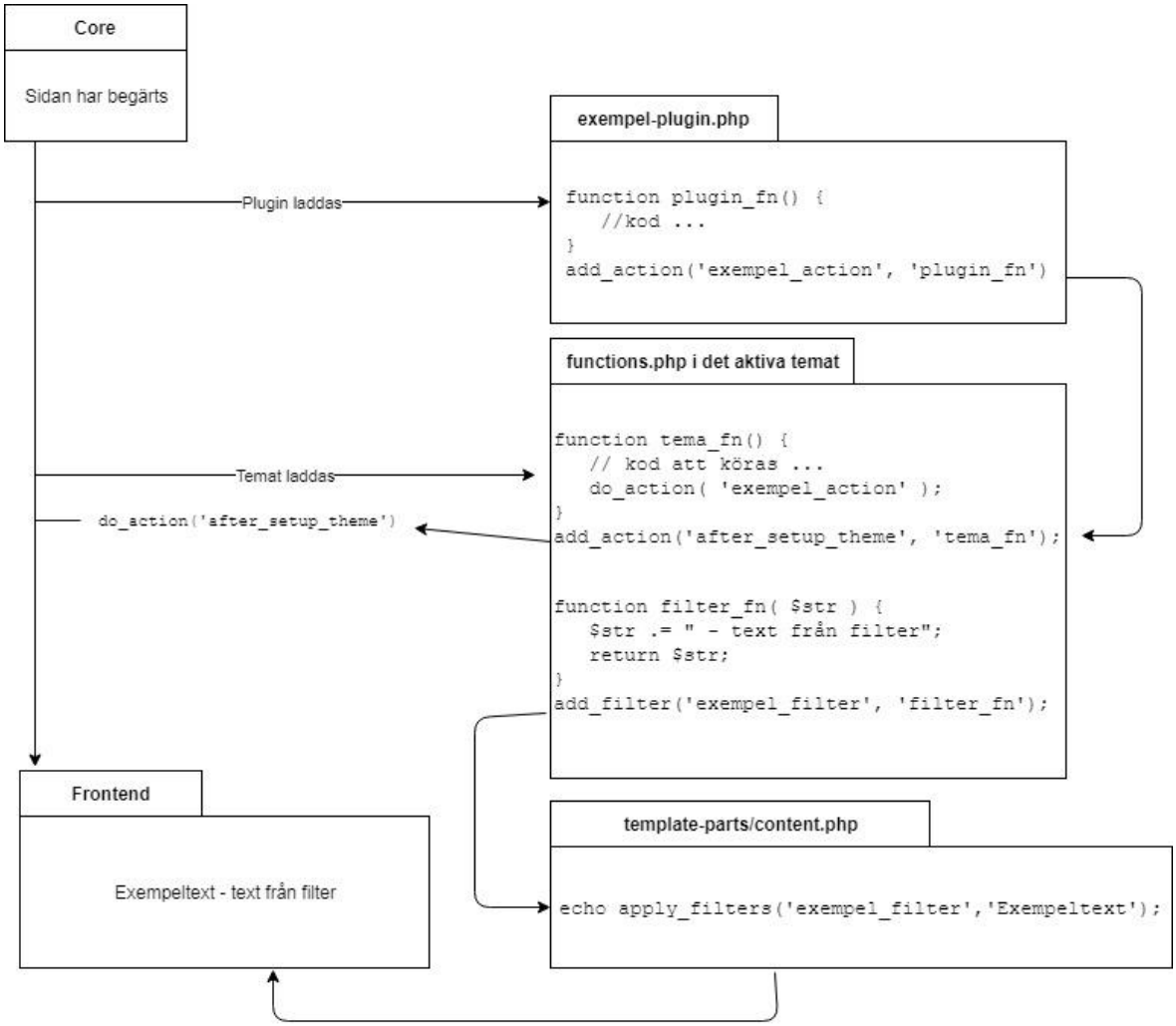
En viktig regel inom WordPress utveckling är att inte modifiera de filer som hör till kärnan. Detta är heller inte nödvändigt tack vare Plugin API.

Plugin API erbjuder funktionaliteten med vilken man bygger ihop de tre byggklossar WordPress består av: kärnan, teman och plugins. Så kallade hooks gör det möjligt för ett tema eller plugin att infoga sin egen kod i, eller modifiera den PHP bearbetning som WordPress utför. Hooks delas in i två kategorier: action hooks och filter hooks (WPC u.å.d).

När en sida i användargränssnittet eller på frontenden laddas, eller när en viss handling såsom skapandet av en ny sida utförs, körs PHP kod. Funktionen `do_action()` skapar actionhooks och finns utspridd i denna kod. Utvecklare kan skriva funktioner, som sedan kan kopplas till dessa actionhooks med hjälp av funktionen `add_action()`. Filter hooks

fungerar ungefär på samma sätt men används för att modifiera innehåll innan det skrivs ut eller förs in i databasen (WPC u.å.d).

Figur 5 beskriver hur WordPress pusslas ihop m.h.a hooks. WordPress kärnan laddar tidigt pluginfilerna. I detta exempel hittas ett aktivt plugin: Exempel plugin, vars huvudfil är exempel-plugin.php. I denna fil definieras en funktion, `plugin_fn()`. I filen finns funktionen `add_action()`, som tar två argument: ”exempel_action” och ”plugin_fn”. Denna funktion används för att koppla `plugin_fn()` till en hook vid namn ”exempel_action”. Detta innebär att när `do_action('exempel_action')` körs, anropas `plugin_fn()`.



Figur 5. Hur Plugin API används för att pussla ihop WordPress

När kärnan laddar det aktiva temat, bearbetas dess functions.php fil. I denna fil finns funktionen `tema_fn()`, som anropas när `do_action('after_setup_theme')` körs. I `tema_fn()` anropas också `do_action('exempel_action')`, som `plugin_fn()` i pluginfilen är kopplad till.

Efter att temat har laddats körs funktionen `do_action('after_setup_theme')` i kärnan. Nu anropar WordPress de funktioner som har kopplats till denna hook, i detta fall `tema_fn()`. I

och med att `tema_fn()` körs, anropas `do_action('exempel_action')`, och i och med att denna funktion körs, anropas `plugin_fn()`.

Jag bör påpeka att detta inte är ett vettigt sätt att använda actionhooks på. Ett realistiskt tillvägagångssätt i detta fall skulle vara att direkt koppla funktionen `plugin_fn()` till ”after_setup_theme”, men för att bättre kunna förklara hur kärnan, teman och plugins kan kopplas ihop valde jag att göra på detta sätt.

I functions.php definieras också en filter funktion, `filter_fn()`. Filterfunktioner tar innehåll som argument och returnerar sedan det modifierade innehållet. Funktionen `add_filter('exempel_filter', 'filter_fn')` lägger till funktionen `filter_fn()` i listan på funktioner som innehåll kopplat till 'exempel_filter' skall gå genom. I template-parts/content.php kan man se hur man sedan tillämpar filtret, och i frontendrutan hur innehållet skrivs ut på frontenden.

4.2 Settings API

WordPress Settings API tillåter skapandet och modifiering av sidor och inställningar i användargränssnittet. En sida i WordPress användargränssnitt består av inmatningselement, som tilldelas sektionstillhörighet. Dessa sektioner tilldelas i sin tur sidtillhörighet (WPC u.å.e).

Funktionerna `add_menu_page()` och `add_submenu_page()` används för att skapa sidor. Dessa funktioner tar ett antal argument, som beskrivs i kodexempel 1. Funktionen `add_submenu_page()` kräver en överordnad sida vars slug anges som första argument. WordPress dokumentation rekommenderar att anropa dessa funktioner vid ”admin_menu” actionhook (WPDR u.å.a).

Kodexempel 1. Skapandet av en inställningssida med Settings APIs add_menu_page() funktion

```
1.  function add_settings_page() {
2.      add_menu_page(
3.        "Theme Settings",//Sidans namn
4.        "Theme Settings",//Sidans namn i menyn
5.        "manage_options",//Rättigheter som krävs för att ha tillgång till sidan
6.        "theme-settings",//Sidans slug, en unik, URL vänlig textsträng
7.        "print_settings_page",//En callback-funktion som skriver ut sidan.
8.      );
9.  }
10. add_action("admin_menu", "add_settings_page");
```

Dessa funktioner tar en callback-funktion som argument. Denna callback-funktion, som kan ses i kodexempel 2, skriver ut innehållet på sidan. Alla inmatningselement bör höra till ett

`<form>` element med action attributet ”options.php”. Funktionen `settings_fields()` skriver ut dolda `<input>` element som WordPress använder för att skydda fält från missbruk. Funktionen `do_settings_section()` skriver ut de sektioner som hör till sidan ”theme-settings”, och `submit_button()` ett `<input type=”submit”>` element (PressCoders (PC) u.å.).

Kodexempel 2. Funktion som skriver ut sida skapad med add_menu_page()

```
1.  function print_settings_page() { ?>
2.      <form method="post" action="options.php">
3.          <?php
4.                  settings_fields("theme-settings");
5.                  do_settings_sections("theme-settings");
6.                  submit_button();
7.              ?>
8.      </form>
9.  <?php }
```

I kodexempel 3 skapas med funktionen `add_settings_section()` en sektion vid namn Settings section, och med funktionen `add_settings_field()` ett inställningsfält som hör till sektionen. Funktionen `register_setting()` registrerar inställningen och dess data, och tilldelar grupptillhörighet, i detta fall till gruppen ”theme-settings” (PC u.å.). Grupptillhörighet bör vara samma för alla inställningar på en sida.

Kodexempel 3. Skapandet av sektioner och fält, samt registrering av inställning med Settings API

```
1.  function create_settings() {
2.      add_settings_section(
3.        "settings_section",//Sektionens slug
4.        "Settings section",//Sektionens titel
5.        "print_section_info",//Funktion som skriver ut information
6.        "theme-settings"//Vilken sida sektionen hör till
7.      );
8.
9.      add_settings_field(
10.       "setting", //Inställningens namn
11.       "Setting",//Inställningens titel
12.       "print_form_element",//Callback-funktion som skriver ut HTML fältet
13.       "theme-settings",//Vilken sida fältet hör till
14.       "settings_section"//Vilken sektion fältet hör till
15.      );
16.
17.      register_setting("theme-settings", "setting");
18. }
19. add_action("admin_init", "create_settings");
```

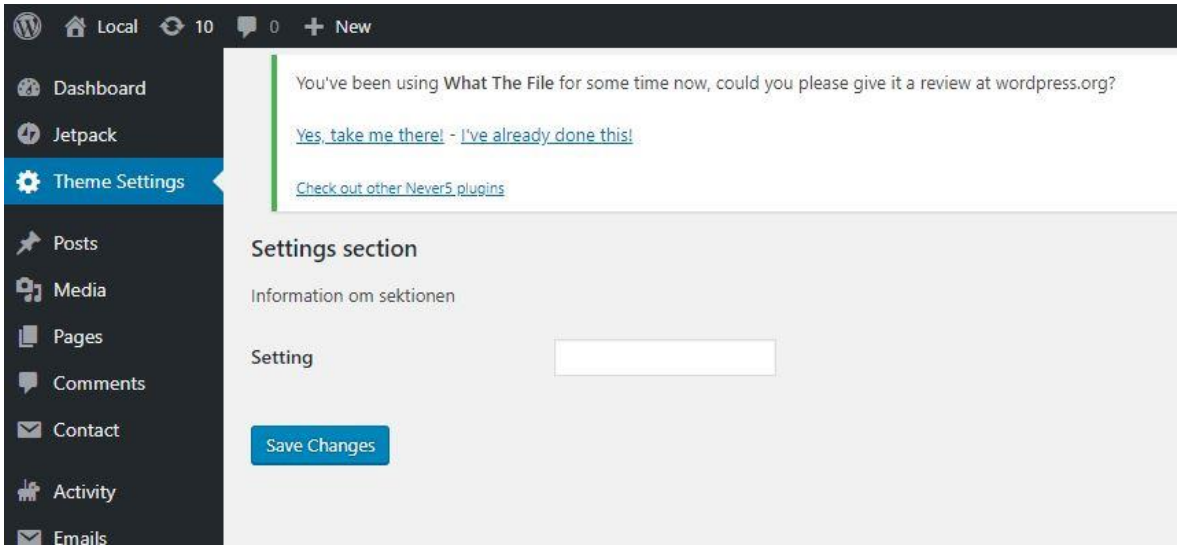
Funktionerna `add_settings_section()` och `add_settings_field()` tar varsin callback-funktion som argument, i detta exempel `print_section_info()` och `print_form_element()`. Dessa funktioner finns med i kodexempel 4. Funktionen `print_section_info()` skriver ut information om sektionen strax under sektionens titel, och funktionen `print_form_element()` inställningen ”setting”:s inmatningselement i HTML. Input elementets name och id attribut

bör vara samma som inställningens slug. Inställningens id, namn och värde lagras som en rad i options tabellen i databasen, och hämtas med funktionen `get_option()`. Här anges värdet som value attribut så att fältet alltid innehåller det nuvarande värdet när sidan laddas.

Kodexempel 4. Funktioner som skriver ut sektionsinformation och ett inmatningselement

```
1.  function print_section_info() {
2.          echo "Information om sektionen";
3.  }
4.
5.  function print_form_element() {
6.      ?>
7.    <input type="text"
            name="setting"
            id="setting"
            value="<?php echo get_option('setting'); ?>" />
8.      <?php
9.  }
```

Kodexemplen i detta kapitel skapar en simpel sida i användargränssnittet, på vilken man har möjlighet att spara en textsträng i databasen (Se Figur 6).



Figur 6. En sida i användargränssnittet skapad med Settings API

4.3 Customize API

Customizer är ett ramverk för förhandsvisning av ändringar som görs i WordPress, och erbjuder ett gränssnitt för anpassning av olika aspekter av hemsidan. Customizer gränssnittet nås som jag tidigre nämnde via Customize under Appearance i användargränssnittet. Customizer gränssnittet består av en sidopanel med inställningar, och ett iframefönster som visar förhandsvisningen (WPDR u.å.n).

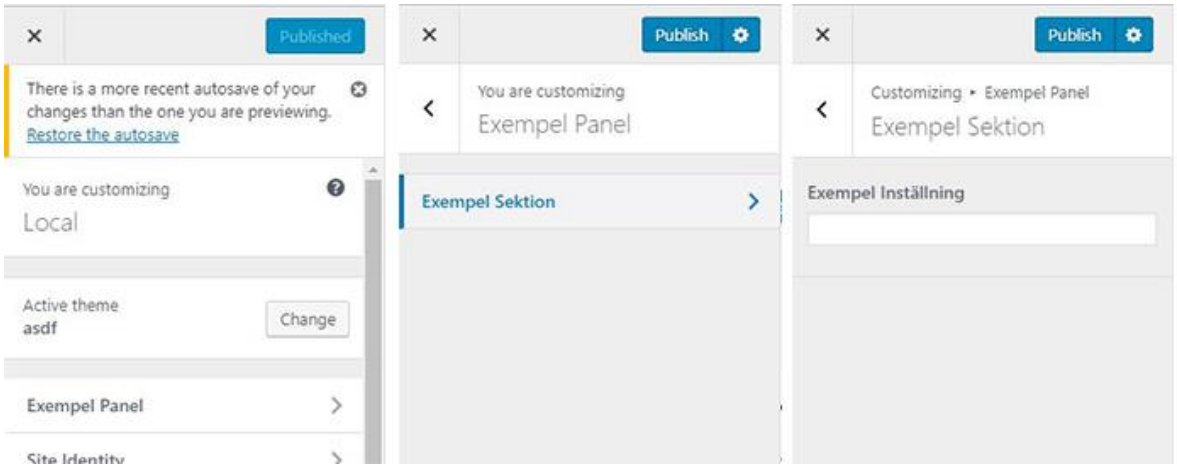
Customize API är objekt orienterat, och består av fyra huvudobjekt: Paneler (panels), sektioner (sections), inställningar (settings) och kontroller (controls). Paneler innehåller en eller flera sektioner, och sektioner innehåller inställningar, som använder sig av kontroller för att associera inmatningselement med värden lagrade i databasen. Customize API erbjuder `add_`, `get_` och `remove_` metoder. Exempelvis används metoden `add_section()` för att skapa en ny sektion, `get_section()` för att hämta en existerande sektion med syftet att sedan ändra dess egenskaper, och `remove_section()` för avaktivera en sektion (WPDR u.å.d).

Tillgång till Customize API:s huvudobjekt, `WP_Customize_Manager`, fås via "customize_register" actionhook (Se Kodexempel 5). I detta exempel skapas först en panel med id "exempel-panel" genom att anropa metoden `add_panel()` på `WP_Customize_Manager` objektet. Liknande metoder används för att skapa en sektion, en inställning och en kontroll. Dessa `add_` metoder tar förutom ett id också ett argument i form av en associative array innehållande olika egenskaper. Värdet för "panel" i `add_section()` avgör vilken panel sektionen hör till, och värdet för "section" i `add_control()` vilken sektion inmatningselementet hör till. Förutom dessa anges i exemplet också titlar för de olika elementen.

Kodexempel 5. Skapandet av Customize APIs fyra komponenter

```
1.  add_action('customize_register','my_customize_register');
2.  function my_customize_register( $wp_customize ) {
3.
4.    $wp_customize->add_panel( 'exempel-panel', array(
5.      'title' => 'Exempel Panel' ));
6.
7.    $wp_customize->add_section( 'exempel-section', array(
8.      'title' => 'Exempel Sektion',
9.      'panel' => 'exempel-panel' ));
10.
11.   $wp_customize->add_setting( 'exempel-setting' );
12.
13.   $wp_customize->add_control( 'exempel-setting', array(
14.     'label'   => 'Exempel Inställning',
15.     'section' => 'exempel-section' ));
16. }
```

Figur 7 är en skärmdump av panelen, sektionen och inställningskontrollern som skapas i kodexempel 5. WordPress kärnan skapar ett antal sektioner i Customizer som innehåller inställningar för bl.a. sidans titel och slogan (WPC u.å.g). Också inställningar gällande logotyp, bild i sidhuvud och bakgrundsbild skapas automatiskt förutsatt att det aktiva temat stöder dessa. Stöd för dessa aktiveras med funktionen `add_theme_support()`. Det bör påpekas att en sektion inte nödvändigtvis bör finnas inuti en panel. Ifall värdet för ”panel” i `add_section()` lämnas tomt befinner sig sektionen högst uppe i hierarkin.



Figur 7. En panel, sektion och inställning skapad med Customize API

Värden för inställningar skapade med Customizer API lagras i databasens options tabell, men till skillnad från inställningar skapade med Settings API lagras alla inställningar i en enda cell (McFarlin 2015). Ett värde hämtas från databasen med funktionen `get_theme_mod()`.

4.3.1 Customizer JavaScript API

JavaScript (JQuery) används i Customizer exempelvis för att dynamiskt dölja eller visa inställningselement eller ändra sidan som förhandsvisas. Varje Customizer PHP objekt har ett motsvarande objekt i JavaScript, som läggs till i det globala objektet `wp` när Customizer gränssnittet laddas (WPDR u.å.m).

Kodexempel 6 beskriver hur man kan dölja ett element baserat på om ett checkboxelement är ikryssat eller inte. Metoden `customize()` kopplar en anonym funktion till inställningen ”test-setting1”. I denna funktion lagras inställningen ”test-setting2”:s HTML container i variabeln `container`. Ifall ”test-setting1” saknar värde (oikryssad) returnerar `value.get()` `false` och metoden `slideUp()` anropas på container elementet. `slideUp()` och `slideDown()` är JQuery metoder som används för att dölja och visa element genom att låta dem glida upp

och ner. `value.bind()` håller koll på inmatningselementet. Den anonyma funktionen körs när inmatningselementets värde ändrar. I funktionen anropas `slideUp()` ifall ett värde finns(Ikryssad), men annars `slideDown()`.

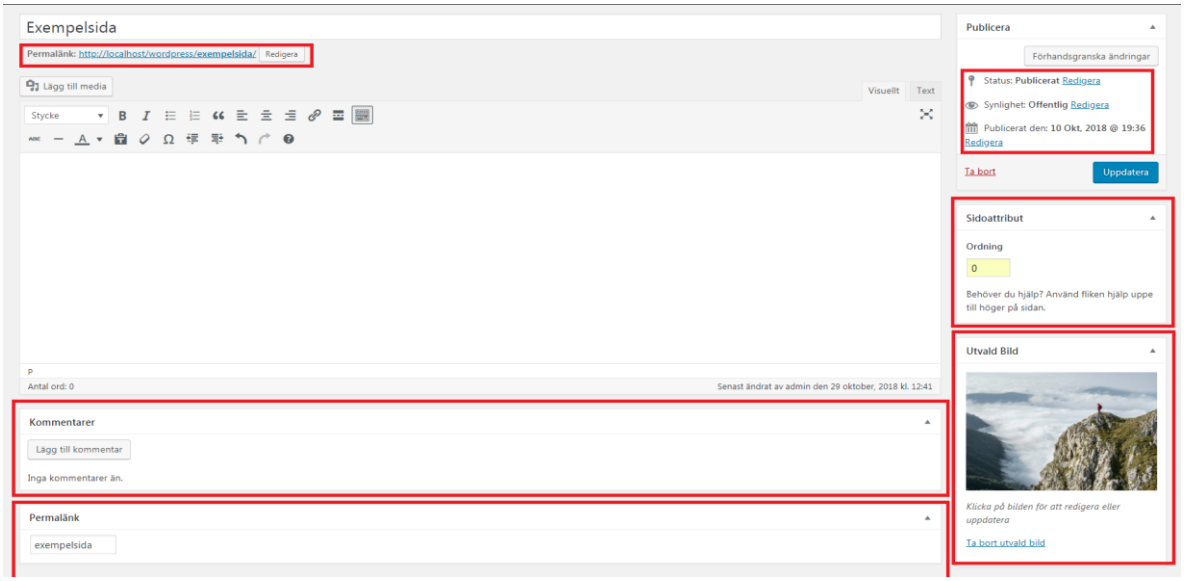
Kodexempel 6. Exempel på användning av Customizer JavaScript API

```
1.  wp.customize( 'test-setting1', function( value ) {
2.          container = wp.customize.control( 'test-setting2' ).container;
3.           if ( !value.get() ) {
4.             container.slideUp(180);
5.           }
6.           value.bind( function( to ) {
7.             if ( to ) { container.slideDown(180); }
8.             else      { container.slideUp(180); }
9.           });
10.       });
```

4.4 Metadata API

Metadata är i WordPress information om inlägg, användare, taxonomier (kategorier, taggar) och kommentarer, och inkluderar exempelvis kategoritillhörighet, skribent, inläggsdatum för inlägg och exempelvis profilbild för användare. Metadata API används för att hämta och manipulera denna metadata, med hjälp av ett antal `add_`, `delete_`, `get_` och `update_` funktioner (McCollin 2014).

Metaboxar kan ses i redigeringsvyn, och innehåller inmatningselement i vilka metadata kan anges. Kärnan skapar för de inbyggda inläggstyperna metaboxar för bl.a. kategoritillhörighet (Se Figur 8).



Figur 8. Metaboxar i redigeringsvyn

Funktionen `add_meta_box()` används i kodexempel 7 för att skapa en metabox som visas när ett inlägg av typen page redigeras. Funktionen `exempel_meta_cb_fn()` skriver ut inmatningselementets HTML. I denna funktion initieras det globala objektet `$post`. Detta objekt innehåller information om det aktuella inlägget i WordPress loopen. Funktionen `get_post_meta()` används här för att hämta data som är kopplat till det aktuella inläggets id. Metanyckeln ”_exempel_meta_key” är namnet på metadatan som ska hämtas. Det sista argumentet, `true`, avgör ifall funktionen returnerar värdet som en array eller som en sträng.

Kodexempel 7. Skapandet av en metabox

```
1.  add_action('add_meta_boxes', 'exempel_metabox_fn');
2.  function exempel_metabox_fn() {
3.    add_meta_box(
4.      'exempel_meta', //ID, bör användas som inmatningselementets name attribut
5.      'Exempel meta box', //Titel, syns ovanför inmatningselementet
6.      'exempel_meta_cb_fn', //Callback-funktion
7.      'page' //sträng eller array med inläggstyper som stöder metaboxen
8.    );
9.  }
10.
11. function exempel_meta_cb_fn() {
12.   global $post;
13.   $value = get_post_meta( $post->ID, '_exempel_meta_key', true );
14.   echo '<input type="text" name="exempel_meta" value="'.$value.' "/>
15. }
```

`update_` funktioner används för att uppdatera metadata tabellerna i databasen. I kodexempel 8 är funktionen `save_postdata()` kopplad till ”save_data” actionhook, som körs när ett inlägg sparas. Argumentet `$post_id` som skickas till funktionen är det sparade inläggets id. `$_POST` är en superglobal som används för att lagra formulärdata som skickas med HTTP POST metoden i PHP, och är en associative array i vilken värdena ser ut på följande sätt: `’inmatningselementets_name_attribut’ => ’elementets innehåll’`. I `save_postdata()` kontrolleras att `$_POST` innehåller nyckeln ”exempel_meta”, och funktionen `update_post_meta()` uppdaterar sedan värdet för metanyckeln ”_exempel_meta_key” med värdet för `$_POST[’exempel_meta’]`. Ifall metanyckeln inte hittas för det aktuella inlägget skapas en ny rad i databasen (WPDR u.å.c).

Kodexempel 8. Uppdatering av metadata

```
1.  add_action('save_post', 'save_postdata');
2.  function save_postdata($post_id) {
3.      if (array_key_exists('exempel_meta', $_POST)) {
4.          update_post_meta(
5.              $post_id,
6.              '_exempel_meta_key',
7.              $_POST[’exempel_meta’]
8.          );
9.      }
10. }
```

Metadata lagras i databasens commentmeta, postmeta, termmeta och usermeta tabeller (McCollin 2014).

4.5 Widgets API

Widgets används i WordPress för att lägga till innehåll och funktionalitet i olika sektioner av hemsidan, i så kallade widget-ready areas, eller widgetområden. Den vanligaste placeringen av dessa områden är i sidofältet och i sidfoten (WPC u.å.k).

Funktionen `register_sidebar()` används för att skapa ett widgetområde (WPDR u.å.g). Denna funktion tar en associative array innehållande namn, id mm. som argument, och bör anropas vid ”widgets_init” actionhook. På Widgets-sidan under Appearance-fliken i användargränssnittet kan man se temats aktiva widgetområden, i form av lådor till vilka man med ett drag & drop system kan dra tillgängliga widgets. Funktionen `dynamic_sidebar()` används i mallfilerna, och skriver ut widgets som har lagts till i ett widgetområde (WPDR u.å.e).

Widgets API erbjuder möjligheten att skapa widgets genom att skriva en PHP klass som förlänger klassen `WP_Widget`, som finns i kärnan. Denna klass bör innehålla fyra metoder: `__construct()`, `widget()`, `form()` och `update()` (WPC u.å.i).

Konstruktormetoden `__construct()` i kodexempel 9 körs när widgetobjektet skapas, och i denna anropas förälderklassen `WP_Widgets` konstruktormetod med widgetens id, namn och en array innehållande bl.a. en beskrivning som argument.

Kodexempel 9. konstruktormetod för WP_Widget subklass

```
1.  public function __construct() {
2.          parent::__construct(
3.              'exempel_widget', // ID
4.              'Exempel Widget', // Namn
5.              array(
6.                  'description' => 'Exempel Widget Beskrivning' // Beskrivning
7.            ));
8.      }
```

Metoden `form()` skriver ut `<form>` elementet som används för att ange widgetinnehåll. Argumentet `$instance` i kodexempel 10 är en array som innehåller den aktuella widgetinstansens innehåll. I detta exempel kontrolleras ifall `$instance['widget_text']` har ett värde som kan användas som inmatningselementets value attribut. Ifall inte får variabeln `$widget_text` värdet ’’. Metoden `get_field_name()` används för att konstruera ett passande

name attribut för inmatningselement inuti `form()` metoden, och skapar här ett name attribut som baserar sig på textsträngen ’widget_text’. Detta krävs eftersom inmatningselement i varje instans av en widget måste ha unika name attribut. All data som anges i dessa inmatningselement lagras i en cell i options tabellen i databasen (WPC u.å.i).

Kodexempel 10. WP_Widget form() metod

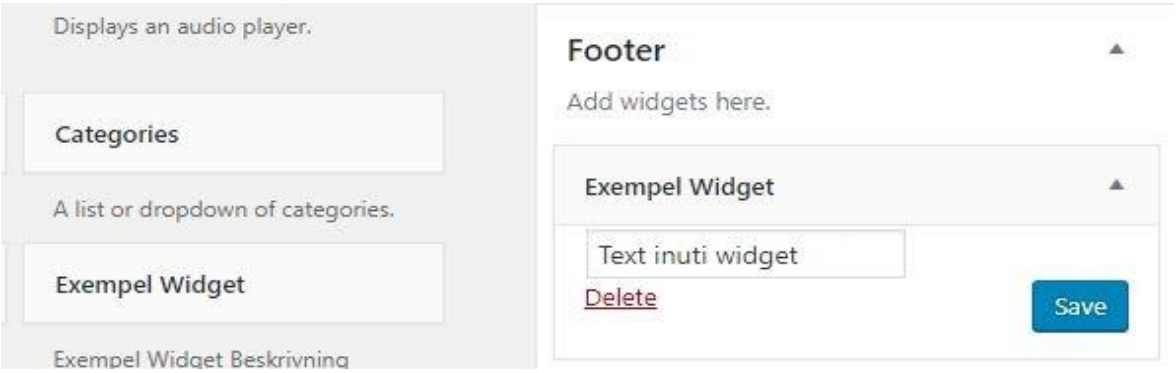
```
1.  public function form( $instance ) {
2.    $widget_text = !empty($instance['widget_text']) ? $instance['widget_text'] : '';
3.    $field_name = $this->get_field_name( 'widget_text' );
4.    echo "<input type='text' name='$field_name' value='$widget_text'>";
5.  }
```

Metoden `update()` (Kodexempel 11) anropas när formulärdatan skickas, och används för att uppdatera datan i `$instance`. Datan från inmatningselementen lagras i `$new_instance`. Värdet för `$instance[’widget_text’]` uppdateras med data som angivits i inmatningselementet. Det uppdaterade värdet för `$instance[’widget_text’]` används nu som value attribut för inmatningselementet i `form()` (WPC u.å.i).

Kodexempel 11. WP_Widget update() metod

```
1.  public function update( $new_instance ) {
2.        $instance = array();
3.        $instance['widget_text'] = !empty($new_instance['widget_text']) ?
4.                                    $new_instance['widget_text'] : '';
5.        return $instance;
6.    }
```

Widgets-sidan I användargränssnittet består av en lista på tillgängliga widgets, och en lista på aktiva widgetområden (Figur 9). Widgetområdet Footer innehåller en instans av den widget som skapats i de senaste kodexemplen. Värdet för `$instance[’widget_text’]` är ”Text inuti widget”.



Figur 9. Initiering av en exempelwidget

Metoden `widget()` (Kodexempel 12) skriver ut widgetinnehållet på frontenden. `$args` är en array som anges som argument för funktionen `register_sidebar()`. Värdet för `$args['before_widget']` och `$args['after_widget']` är ofta HTML, som används exempelvis för att omsluta alla widgets i widgetområdet med element av samma klass (WPC u.å.i).

Kodexempel 12. WP_Widget widget() metod

```
1.  public function widget( $args, $instance ) {
2.          echo $args['before_widget'];
3.      echo $instance['widget_text'];
4.          echo $args['after_widget'];
5.      }
```

4.6 REST API

WordPress REST API gör det möjligt att med HTTP-begäran hantera WordPress innehåll genom att skicka och begära data i JSON format. Detta gör det exempelvis möjligt att externt interagera med WordPress, eller att ladda innehåll från databasen utan siduppdatering med AJAX (WPDR u.å.h).

REST API består av URLer, eller routes, till vilka HTTP-begäran skickas. API endpoints är CRUD metoder som används för att skapa (Create), läsa (Read), uppdatera (Update) och radera (Delete) WordPress data (Shahid 2016).

Som ett exempel, ifall man skickar en GET-begäran till routen /wp/v2/posts returneras information om inlägg av typen post. Denna information är bl.a. inläggens id, skribent, slug, titel och innehåll. Med en POST-begäran till samma route har man möjlighet att skicka JSON data innehållande inläggsinformation, och på så sätt skapa ett nytt inlägg. GET-begäran till routen anropar en `get_item` metod som returnerar inlägg, och POST-begäran en `create_item` metod som skapar ett nytt inlägg. Routen /wp/v2/posts har alltså två endpoints. Roten för REST API är alltid /wp-json/, och ifall man skickar en GET begäran till denna URL får man alla tillgängliga routes som svar (WPDR u.å.h).

Det är möjligt att skapa egna REST routes med funktionen `register_rest_route()`, (Kodexempel 13). Denna funktion bör kopplas till ”rest_api_init” actionhook. Den första delen av routen anges som funktionens första argument, och routens bas URL som det andra argumentet. `(?P<id>\d+)` är ett regex uttryck som används för att skapa ett heltalsvärde med nyckeln ”id” i en associative array som skickas till callback-funktionen `exempel_endpoint_fn()`. I `register_rest_route()` anges en array som tredje argument,

innehållande ”methods” och ”callback”. Värdet för ”methods” avgör routens endpoints, och ”callback” den funktion som anropas när en HTTP-begäran gjorts till routen (WPDR u.å.i).

Kodexempel 13. Skapandet av en REST API route

```
1.  function register_exempel_route() {
2.      register_rest_route( 'exempel-route/v1', '/exempel-bas/(?P<id>\d+)', array(
3.          'methods'  => 'GET',
4.          'callback' => 'exempel_endpoint_fn',
5.      ));
6.  }
7.  function exempel_endpoint_fn( $data ) {
8.      return $data['id'];
9.  }
```

Funktionerna i kodexempel 13 skapar en route som nås på följande sätt: http://exempel.fi/wp-json/exempel-route/v1/exempel-bas/6. Denna route har en endpoint. När en GET-begäran skickas till denna URL returneras `$data['id']` vilket i detta exempel är 6 .

4.7 Shortcode API

En shortcode, eller kortkod, är en kodsträng som kan anges i TinyMCE för att anropa PHP funktioner. En kortkod kan i sin simplaste form se ut på följande sätt: `[kortkod]`, men det är möjligt för kortkoder att acceptera attribut och innehåll: `[kortkod attribut="a"]Innehåll[/kortkod]` (WPC u.å.f).

När `the_content()`, en template tag som skriver ut ett inläggs innehåll, anropas i WordPress loopen analyseras de kortkoder som hittas i texten, och skickar dess attribut och innehåll till den funktion som kortkoden kopplats till. Denna funktion bör returnera ett värde, som sedan skrivs ut istället för själva kortkoden (WPC u.å.f).

WordPress kärnan erbjuder ett antal kortkoder, som används för att bl.a. bädda in mediafiler. Med hjälp av Shortcode API är det möjligt att utöka listan med tillgängliga kortkoder. Funktionen `add_shortcode()` används för att koppla en kortkod till en PHP funktion (Kodexempel 14). Till callback-funktionen `exempel_shortcode_fn()` skickas tre argument: en associative array innehållande angivna attribut, en sträng innehållande angivet innehåll, och en sträng innehållande kortkodens namn. Funktionen `shortcode_atts()` används för att ange standardvärden för attribut i `$atts` (WPC u.å.f). Kortkoden i detta exempel kunde användas på följande sätt: `[exempel-kortkod namn="Namn"]Innehåll[/exempel-kortkod]`, och skulle då returnera strängen ”Namn, Innehåll, exempel-kortkod”.

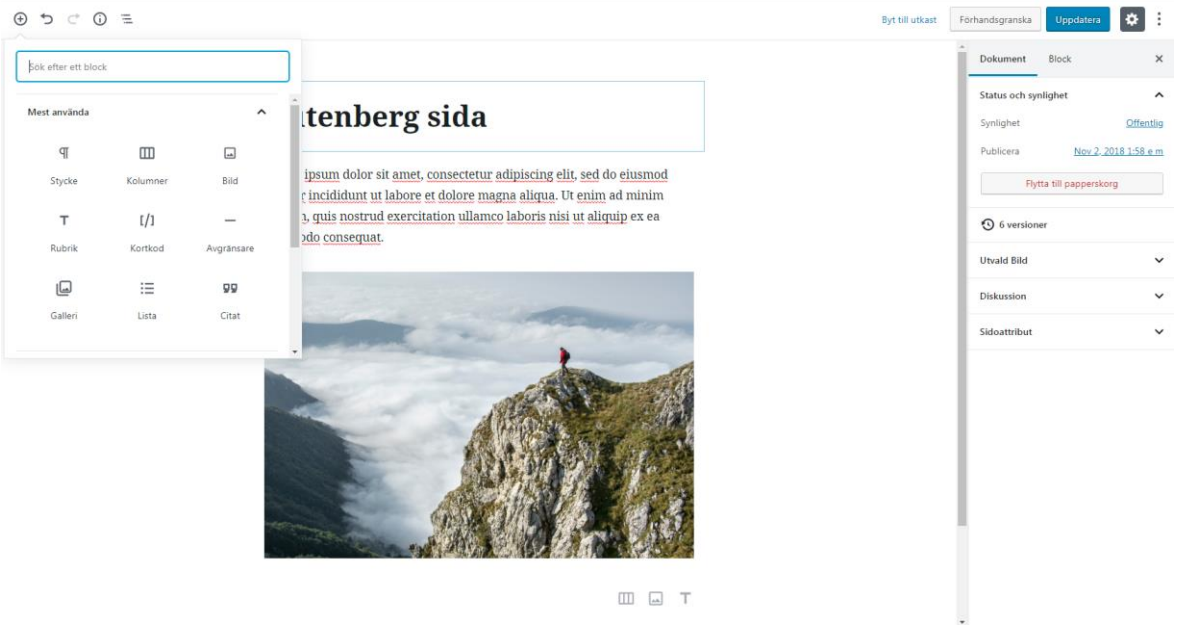
Kodexempel 14. Skapandet av en kortkod

```
1.  function exempel_kortkod_fn( $atts, $content='', $tag ) {
2.    $a = shortcode_atts( array(
3.      'namn' => 'Namn'
4.    ), $atts );
5.    return $a['namn'] . ', ' . $content . ', ' . $tag;
6.  }
7.  add_shortcode( 'exempel-kortkod', 'exempel_kortkod_fn' );
```

5 Gutenberg

Gutenberg är benämningen på den nya redigeringsupplevelsen som utvecklas av WordPress. Projektet har tre planerade skeden, varav det första involverar en ombyggnad av inläggsredigeraren. Denna redigeraren, som ersätter TinyMCE, är för tillfället tillgängligt via ett plugin, men kommer i WordPress version 5 släppas som en del av kärnan. Version 5 kommer ut i slutet av 2018. Under 2019 kommer det andra skedet att fokusera på sidmallar, och det tredje på fullständiga sidanpassningsmöjligheter (Gutenberg readme u.å).

Gutenbergs inläggsredigerare baserar sig på modulära block, som representerar olika sorters innehåll, såsom rubriker, paragrafer, bilder mm. Dessa block läggs till via en meny innehållande tillgängliga blocktyper, och kan sedan redigeras och flyttas på m.h.a. ett drag & drop system (Figur 10).



Figur 10. Gutenbergs redigeringsgränssnitt

5.1 Block API

Gutenberg redigeraren är byggd med React, ett JavaScript bibliotek utvecklat av Facebook och Instagram. React används för att skapa användargränssnitt, och baserar sig på komponenter som tillsammans utgör gränssnittet (React u.å.b).

Block API används för att skapa egna blocktyper, och består av ett antal objekt som läggs till i det globala `wp` objektet. För tillfället är Block API endast tillgänglig i inläggsredigeringsgränssnittet. Metoden som används för att skapa blocks, `registerBlockType()`, finns tillsammans med övriga blockhanteringsmetoder i `wp.blocks` objektet. Det rekommenderas att ladda JavaScriptfiler som registrerar blocks vid ”enqueue_block_editor_assets” actionhook, eftersom denna hook endast körs när Gutenberg gränssnittet laddas (Schofield 2018).

I följande kodexempel används en JavaScript preprocessor vid namn JSX. JSX möjliggör kod som kan beskrivas som en blandning av HTML och JavaScript. Denna kod kompileras sedan till webbläsargiltig JavaScript (Agero 2017). Dokumentationen för React (u.å.a) rekommenderar användning av JSX för enklare byggande av användargränssnitt.

Metoden `registerBlockType()` tar två argument, ett namn och ett konfigurationsobjekt (Se kodexempel 15). Blockets namn bör struktureras på följande sätt: namn-på-temat-eller-plugin/blockets-namn. Konfigurationsobjektet kan innehålla ett antal attribut, såsom titel, beskrivning, blockkategori mm. Objektet bör innehålla två metoder vid namn `edit()` och `save()`. Metoden `edit()` avgör vad blocket skriver ut i användargränssnittet, och `save()` vad som skrivs ut på frontenden (Schofield 2018).

Kodexempel 15. Skapandet av ett Gutenbergblock

```
1.  wp.blocks.registerBlockType( 'plugin-namn/block-test-block', {
2.      title: 'test-block',//Blocktypens titel
3.      category: 'common',//Var blocktypen kan hittas i menyn
4.      edit: function() {
5.          return ( <p>Detta syns i redigeringsgränssnittet</p> );
6.      },
7.      save: function() {
8.          return ( <p>Detta syns på frontend</p> );
9.      },
10. });
```

Blocket i kodexempel 15 skriver ut statisk html både i redigeringsgränssnittet och på frontenden. För att möjliggöra dynamiskt innehåll används i Gutenberg ett `attributes` objekt, som läggs till i konfigurationsobjektet.

Kodexempel 16. ett Gutenbergblocks attributes objekt

```
1.  attributes: {
2.      exempelText: {
3.              type: 'string',
4.              selector: '.text'
5.              source: 'text',
6.
7.      }
8.  }
```

I kodexempel 16 läggs ett blockattribut vid namn `exempelText` till i `attributes` objektet. Värdet för `type` avgör blockattributets datatyp och värdet för `selector` vilket HTML element som attributvärdet ska hämtas från. När sidan laddas på frontenden skrivs HTML som hämtas från databasens post_content kolumn ut. Denna HTML hämtas också när redigeringsgränssnittet laddas, men istället för att direkt skrivas ut konverteras den till Gutenberg block. Under denna process tilldelas blockattributet `exempelText` ett värde som hittas i ett HTML element av klassen 'text'. `Source` avgör vilken data som skall hämtas. I kodexempel 16 hämtas den text som finns mellan HTML taggarna, men för att exempelvis hämta ett elements klassnamn kan `source: 'attribute', attribute: 'class'` anges. Det krävs inte att ett blockattribut innehåller värden för `source` och `selector`. Ifall dessa inte anges sparas och läses blockattributet från blockavgränsarkommentaren. Blockavgränsarkommentarer används i post_contents HTML för att separera block och lagra blockinformation (WordPress (WP) u.å.c).

För att manipulera blockattribut används metoden `setAttributes()`. Denna metod hittas tillsammans med blocktypens namn, beskrivning, blockattribut mm. i ett objekt som heter `props` (WP u.å.a). Detta objekt skickas till `edit()` och `save()` metoderna. Kodexempel 17 visar en uppdaterad version av `edit()` metoden. I gränssnittet skrivs nu ett inmatningselement med värdet för `exempelText` som value attribut ut. onChange attributet ser till att funktionen `onTextChange()` körs varje gång elementet ändras. Till denna funktion skickas ett objekt som ger information om inmatningselementet i fråga, och blockattributet `exempelText` uppdateras med inmatningselementets value attribut.

Kodexempel 17. Gutenberg edit() metod som tillåter modifiering av blockattribut

```
1.  edit: function() {
2.    function onTextChange(changes) {
3.                props.setAttributes({
4.                    exempelText: changes.target.value
5.                });
6.            }
7.    return (
8.  <input type="text" value={props.attributes.exempelText} onChange={onTextChange}/>
9.     );
10. }
```

Till databasen skickas det innehåll som metoden `save()` returnerar, och denna metod avgör därmed vad som visas på frontenden (WP u.å.b). För att spara värdet för `exempelText` kan metoden uppdateras att returnera följande:

```
<p className="text"> { props.attributes.exempelText } </p>
```

När metoden körs sparas följande HTML till post_content, där ’Värdet’ är värdet för `exempelText`.:

```
<!-- wp:plugin-namn/block-test-block -->
<div class="wp-block-cgb-block-test-block"><p class="text">Värdet</p></div>
<!-- /wp: plugin-namn/block-test-block -->
```

Alla block är omslutna av dessa avgränsarkommentarer, som innehåller åtminstone blocktypens namn. Det krävs som sagt inte att ett blockattribut innehåller värden för `source` och `selector`, och ifall detta hade varit fallet i detta exempel skulle den öppnande kommentaren se ut på följande sätt:

```
<!-- wp:cgb/block-test-block {"exempelText":"Värde"} -->
```

6 Avslutning

I detta arbete har jag presenterat innehållshanteringssystemet WordPress. Jag har beskrivit hur systemets olika delar samverkar och hur de webbsidor som besökaren ser produceras. Jag har också förklarat hur en del av de APIs som WordPress erbjuder fungerar och gått grundligt in på API funktionernas användning. WordPress framtid, Gutenberg, har också presenterats.

WordPress är ett ganska brett ämne, och ett problem som uppstod under projektet var att jag på grund av dålig planering inte riktigt visste hur jag ville avgränsa arbetet. Detta ledde till

en del ändringar i arbetets utformning och kapitelindelning. En del text togs också bort i samband med att jag ändrade arbetets fokus.

Syftet med detta arbete var att utveckla mina kunskaper inom WordPress-utveckling. Detta tycker jag trots dålig planering att jag har lyckats bra med och jag känner mig mycket mindre begränsad vad gäller utveckling av teman och plugins. Detta eftersom jag nu har en relativt bra förståelse över hur systemet fungerar, och hur de olika APIs kan användas.

Källförteckning

Agero, 2017. *Kickstarta din Front-end med React - Del 2: React och JSX* [Online] https://www.agero.se/blogg/kickastarta-din-reactjs-utveckling-del2 [Hämtat: 03.11.2018]

Ewer, T., 2016. *Understanding the WordPress File and Directory Structure* [Online] https://www.elegantthemes.com/blog/tips-tricks/understanding-the-wordpress-file-and-directory-structure [Hämtat: 04.09.2018]

Gutenberg readme, (u.å.). *Gutenberg* [Online] https://github.com/WordPress/gutenberg/blob/master/README.md [Hämtat: 01.11.2018]

iThemes, (u.å.). *What is Wordpress?* [Online] https://ithemes.com/tutorials/what-is-wordpress/ [Hämtat: 15.11.2018].

McCollin, R., 2014. *Understanding and Working with Metadata in WordPress* [Online] https://code.tutsplus.com/tutorials/understanding-and-working-with-metadata-in-wordpress--cms-21034 [Hämtat: 20.09.2018]

McFarlin, T., 2015. *WordPress Options and Theme Modifications* [Online] https://tommcfarlin.com/wordpress-options/ [Hämtat: 20.10.2018]

Muldoon, K., 2014. *The WordPress Loop Explained For Beginners* [Online] https://www.elegantthemes.com/blog/tips-tricks/the-wordpress-loop-explained-for-beginners [Hämtat: 11.10.2018]

Pataki, D., 2011. *WordPress Essentials: How To Create A WordPress Plugin* [Online] https://www.smashingmagazine.com/2011/09/how-to-create-a-wordpress-plugin/ [Hämtat: 08.09.2018]

Pataki, D., 2017. *WordPress Development for Beginners: Getting Started* [Online] https://premium.wpmudev.org/blog/wordpress-development-beginners-getting-started/ [Hämtat: 04.09.2018]

PressCoders, (u.å.). *Settings API Explained* [Online] http://www.presscoders.com/wordpress-settings-api-explained/ [Hämtat: 02.09.2018]

React, (u.å.a). *Introducing JSX* [Online] https://reactjs.org/docs/introducing-jsx.html [Hämtat: 07.11.2018]

React, (u.å.b). *React - A JavaScript library for building user interfaces* [Online] https://reactjs.org/ [Hämtat: 03.11.2018]

Schofield, J., 2018. *WordPress Gutenberg Blocks Made Easy* [Online] http://jschof.com/gutenberg-blocks/gutenberg-blocks-made-easy/ [Hämtat: 03.11.2018]

Shahid, B., 2016. *WP REST API: Creating, Updating, and Deleting Data* [Online] https://code.tutsplus.com/tutorials/wp-rest-api-creating-updating-and-deleting-data--cms-24883 [Hämtat: 03.10.2018]

WhoIsHostingThis, (u.å.). *The 60 Second Guide to B2/Cafelog* [Online] https://www.whoishostingthis.com/resources/b2-cafelog/ [Hämtat: 11.04.2018].

Vining, R., 2008. *Static HTML Web Pages vs. Content Management Systems* [Online] https://www.robertswebdesign.com/blog/website-design/12-static-html-web-pages-vs-content-management-systems [Hämtat: 10.04.2018].

WordPress, (u.å.a). *Attributes* [Online] https://wordpress.org/gutenberg/handbook/block-api/attributes/ [Hämtat: 07.11.2018]

WordPress, (u.å.b). *Edit and Save* [Online] https://wordpress.org/gutenberg/handbook/block-api/block-edit-save/ [Hämtat: 07.11.2018]

WordPress, (u.å.c). *The Language of Gutenberg* [Online] https://wordpress.org/gutenberg/handbook/language/ [Hämtat: 07.11.2018]

WordPress Codex, (u.å.a). *TinyMCE* [Online] https://codex.wordpress.org/TinyMCE [Hämtat: 10.04.2018].

WordPress Codex, (u.å.b). *Child Themes* [Online] https://codex.wordpress.org/Child_Themes [Hämtat: 07.09.2018]

WordPress Codex, (u.å.c). *Functions File Explained* [Online] https://codex.wordpress.org/Functions_File_Explained [Hämtat: 07.09.2018]

WordPress Codex, (u.å.d). *Plugin API* [Online] https://codex.wordpress.org/Plugin_API [Hämtat: 08.09.2018]

WordPress Codex, (u.å.e). *Settings API* [Online]
https://codex.wordpress.org/Settings_API [Hämtat: 02.09.2018]

WordPress Codex, (u.å.f). *Shortcode API* [Online]
https://codex.wordpress.org/Shortcode_API [Hämtat: 10.10.2018]

WordPress Codex, (u.å.g). *Theme Customization API* [Online]
https://codex.wordpress.org/Theme_Customization_API [Hämtat: 11.09.2018]

WordPress Codex, (u.å.h). *Using Permalinks* [Online]
https://codex.wordpress.org/Using_Permalinks [Hämtat: 05.09.2018]

WordPress Codex, (u.å.i). *Widgets API* [Online]
https://codex.wordpress.org/Widgets_API [Hämtat: 21.09.2018]

WordPress Codex, (u.å.j). *WordPress APIs* [Online]
https://codex.wordpress.org/WordPress_API%27s [Hämtat: 08.09.2018]

WordPress Codex, (u.å.k). *WordPress Widgets* [Online]
https://codex.wordpress.org/WordPress_Widgets [Hämtat: 21.09.2018]

WordPress Developer Resources, (u.å.a). *add_menu_page()* | *Function* [Online]
https://developer.wordpress.org/reference/functions/add_menu_page/
[Hämtat: 02.09.2018]

WordPress Developer Resources, (u.å.b). *add_theme_support()* | *Function* [Online]
https://developer.wordpress.org/reference/functions/add_theme_support/
[Hämtat: 07.09.2018]

WordPress Developer Resources, (u.å.c). *Custom Meta Boxes* [Online]
https://developer.wordpress.org/plugins/metadata/custom-meta-boxes/
[Hämtat: 11.11.2018]

WordPress Developer Resources, (u.å.d). *Customizer Objects* [Online]
https://developer.wordpress.org/themes/customize-api/customizer-objects/
[Hämtat: 11.09.2018]

WordPress Developer Resources, (u.å.e). *dynamic_sidebar()* | *Function* [Online]
https://developer.wordpress.org/reference/functions/dynamic_sidebar/
[Hämtat: 11.11.2018]

WordPress Developer Resources, (u.å.f). *Main Stylesheet (style.css)* [Online]
https://developer.wordpress.org/themes/basics/template-files/ [Hämtat: 05.09.2018]

WordPress Developer Resources, (u.å.g). *register_sidebar()* | *Function* [Online]
https://developer.wordpress.org/reference/functions/register_sidebar/
[Hämtat: 11.11.2018]

WordPress Developer Resources, (u.å.h). *REST API Handbook* [Online]
https://developer.wordpress.org/rest-api/ [Hämtat: 03.10.2018]

WordPress Developer Resources, (u.å.i). *Routes and Endpoints* [Online]
https://developer.wordpress.org/rest-api/extending-the-rest-api/routes-and-endpoints/
[Hämtat: 03.10.2018]

WordPress Developer Resources, (u.å.j). *Template Files* [Online]
https://developer.wordpress.org/themes/basics/template-files/ [Hämtat: 05.09.2018]

WordPress Developer Resources, (u.å.k). *Template Hierarchy* [Online]
https://developer.wordpress.org/themes/basics/template-hierarchy/ [Hämtat: 04.09.2018]

WordPress Developer Resources, (u.å.l). *Template Tags* [Online]
https://developer.wordpress.org/themes/basics/template-tags/ [Hämtat: 04.09.2018]

WordPress Developer Resources, (u.å.m). *The Customizer JavaScript API* [Online]
https://developer.wordpress.org/themes/customize-api/the-customizer-javascript-api/
[Hämtat: 21.10.2018]

WordPress Developer Resources, (u.å.n). *Theme Options – The Customize API* [Online]
https://developer.wordpress.org/themes/customize-api/
[Hämtat: 11.09.2018]

WordPress Developer Resources, (u.å.o). *What is a Theme?* [Online]
https://developer.wordpress.org/themes/getting-started/what-is-a-theme/
[Hämtat: 12:04.2018]

WordPress Developer Resources, (u.å.p). *wp_enqueue_style()* | *Function* [Online]
https://developer.wordpress.org/reference/functions/wp_enqueue_style/
[Hämtat: 07.09.2018]

WPBeginner, (u.å.a). *What is: Post Slug* [Online]
https://www.wpbeginner.com/glossary/post-slug/ [Hämtat: 05.09.2018]

WPBeginner, (u.å.b). *What is: Widgets* [Online]
https://www.wpbeginner.com/glossary/widgets/ [Hämtat: 12.09.2018]

WPBeginner, 2016. *How to Create Custom Post Types in WordPress* [Online]
https://www.wpbeginner.com/wp-tutorials/how-to-create-custom-post-types-in-wordpress/
[Hämtat: 12.04.2018]

WPBeginner, 2017. *What Are WordPress Plugins? And How Do They Work?* [Online]
http://www.wpbeginner.com/beginners-guide/what-are-wordpress-plugins-how-do-they-work/ [Hämtat: 11.04.2018].

Figurförteckning

Kodförteckning

Bilagor

Bilaga 1. Hur WordPress använder mallhierarkin

URLen http://exempel.com/?exempeltyp=exempel-1 har begärts.

Baserat på querysträngen vet WordPress att söka i ”Singular Page”-trädet och fortsätta i ”Single Post Page”. Detta eftersom inlägget inte är av typen ”Page”.

WordPress avgör att Exempel 1 är en ”Custom post” eftersom inläggstypen Exempeltyp inte är en inbyggd inläggstyp.

Det finns möjlighet att skapa mallfiler och manuellt välja att använda dem i inläggets redigeringsvy. Ifall ingen sådan fil har valts fortsätter WordPress att söka efter en passande mallfil i följande ordning: single-exempeltyp-exempel-1.php > single-exempeltyp.php > single.php > singular.php.

Ifall ingen av dessa filer hittas används den ända mallfil som är absolut nödvändig för att ett tema skall fungera: index.php.

(WordPress Developer Resources, u.å.)