Templating with Handlebars

- In order for a web application to grow to any meaningful size, it becomes necessary to generate content dynamically.

- Dynamically-generated content is HTML content that is created programmatically based on data stored (usually in databases) on the server.

- A Facebook feed is a great example of dynamically generated content. Specifically, everyone's Facebook feed has the same general appearance: it is a list of posts, where each post appears more or less the same, apart from its specific content. However, each individual Facebook feed is unique. My feed contains posts and information relevant to me, and yours contains information relevant to you.

- Of course, Facebook's developers don't spend their time coding billions of static pages, each one of which represents the feed of a different user. Instead, when it comes to feeds, they essentially maintain a single page that specifies the general structure and appearance of a feed, minus the specific content.

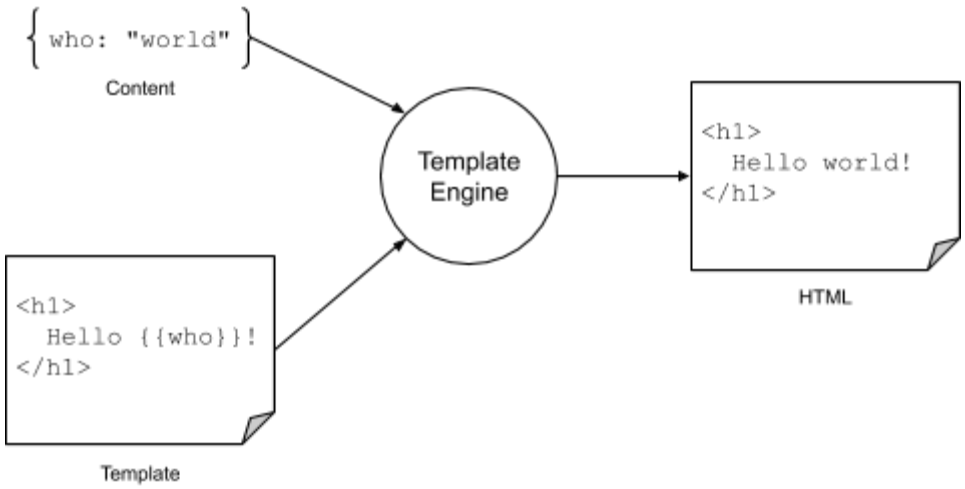
- Each individual Facebook user's feed, then, is generated programmatically whenever that user requests it by grabbing data from the Facebook databases for the user and plugging that data into the feed representation to generate HTML to send back to the user.

- There are different ways of dynamically generating content. Here, we will explore a dynamic content generation approach called ***templating***. A templating system is one that allows us to specify the structure and appearance of a web page (or even a small piece of a webpage, like an individual post on Facebook), leaving placeholders to indicate where actual content should be dynamically inserted.

- When a specific page (or a piece of one) is requested by a client, we programmatically fetch the content for that page and pass that content into our template, which plugs it into the appropriate placeholders to generate actual

HTML content.

- At a high level, this process looks something like this:



- The specific templating engine we'll explore is called Handlebars.

Using Handlebars templates on the client side

- Let's look first at how we can use Handlebars templating to generate content on the client-side.

- For example, let's say we're working on an application where users can see captioned photos of people, something like this:

- Here, each photo is displayed in a card, and each card has the same HTML structure. The only difference between the cards is the URL specifying the photo and the caption text.

- Let's say we want to add functionality on this site to allow the user to click the little red camera button in the corner to enter a new photo URL and an accompanying caption, and then, using the info they enter, a new photo card will be generated and displayed on the page.

- We've already seen how we can do this by using native JavaScript methods to construct a DOM element and then to insert that DOM element into the page. However, this isn't the best practice from a software engineering perspective because it means that we need to implement the photo card structure in our JS code in addition to our HTML code.
    - This means that if we want to change the structure of our photo card, we need to do it in multiple places, which is problematic because it increases the chance for errors and/or inconsistency.

- Once we start using a template engine like Handlebars, we can work towards writing a single implementation of parts of our site that are re-used many times, like our photo card.

- Let's start by creating a Handlebars template to represent our photo card. A Handlebars template looks very much like HTML, except in the places where we want to be able to change the content, the Handlebars template will instead contain *placeholders* into which content will later be inserted.

- Here's what a Handlebars template representing our photo card might look like:

```
<section class="photo-card">
  <div class="img-container">
    <img class="person-photo-img" src="{{photoURL}}" />
  </div>
  <div class="caption">
    {{caption}}
  </div>
</section>
```

- In this template, `{{photoURL}}` and `{{caption}}` are Handlebars expressions indicating locations into which content will be dynamically inserted. Specifically, `photoURL` and `caption` are identifiers that act like parameters to the template. Eventually, we will pass arguments into this template, one of which will be named `photoURL` and one of which will be named `caption`, and the values of these arguments will be inserted into the corresponding locations in the template.
    - Note that in Handlebars, when we enclose an identifier within two sets of curly braces, like `{{photoURL}}` and `{{caption}}`, Handlebars will escape any content that's placed into those locations, which means it's safe to pass user-supplied content directly into such expressions.

- The rest of the template simply specifies the HTML structure of a photo card.

- To be able to use our photo card template to generate new photo cards on the client, there are a few steps we need to follow. First of all, let's put that photo card template above into its own file, say `photoCard.handlebars`.

- Next, we need to transform the template into a form that's usable on the client-side. We'll specifically want to invoke our template from the client-side JS, e.g. in an event handler listening for clicks on a specific button: when the button is clicked, we want to generate a new photo card and insert it into the page.

- With Handlebars, we can actually *compile* our template into a JS function that we can call, passing in as arguments the content we want to insert into the template, and the template function will return an HTML string representing a new photo card.

- We can perform this compilation on the server side and then serve the file containing the compiled JS template function, thus making that function available to be used on the client side.

- To do this, we'll need to install `handlebars` as a dependency to our server-side code:

```
npm install --save handlebars
```

- Once the `handlebars` dependency is installed, the easiest way to use it to compile our template is to write a new script in our `package.json` file.
    - Remember that scripts are simply terminal commands that are commonly used when working with our server-side code.

- Specifically, let's write a script named `build` that performs the compilation:

```
"scripts": {
  ...
  "build": "handlebars path/to/photoCard.handlebars -f
    public/photoCardTemplate.js",
  ...
}
```

- This script runs the Handlebars compiler passing as input the file containing our photo card template and generating as output a new JS file named `public/photoCardTemplate.js`. This file will contain our photo card template function.
    - Here, we're assuming that our server process is already set up to statically serve files located in the directory `public/`.

- To run our build script we can simply use the command:

```
npm run build
```

- Note that we may need to modify our photo card template from time to time. To make sure we're always using the newest version of the template, we can tell `npm` to run our build script before the server starts whenever we run `npm start`. To do this, we can simply add a script named `prestart` that runs our build script:

```
"scripts": {
  ...
  "build": "...",
  "prestart": "npm run build",
  ...
}
```

- With this pre-start script in place, our build script will be run automatically every time we run `npm start` to start our server.

- Once our photo card template is compiled and being served, we can include it in our client-side code, just like we include any other client-side JS script:

```
<script src="/photoCardTemplate.js"></script>
```

    - Here, we're assuming `photoCardTemplate.js` is being served out of our server's root path `/`.

- Importantly, we must include the photo card template script *before* including any scripts in which we'll want to *use* the photo card template. For example, if we wanted to invoke the photo card template from a client-side script named `index.js`, we'd need to include them in this order:

```
<script src="/photoCardTemplate.js"></script>
<script src="/index.js"></script>
```

- We also need to include the Handlebars runtime library in our client-side code, so we can actually invoke our template there. We can use the version of this library that's served on the cdnjs content distribution network:

```
<script
src="https://cdnjs.cloudflare.com/ajax/libs/handlebars.js/4
```

```
.0.11/handlebars.runtime.js"></script>
```

- With everything now set up, we can start to use our photo card template on the client side. In our client-side JS, our template will be available as a function called `Handlebars.templates.photoCard()`.
    - Here, the `photoCard` portion of the function name is derived from the name of the file in which the template was originally implemented: `photoCard.handlebars`.
    - `Handlebars.templates` is a globally-available object in our client-side JS code that is provided by our inclusion of the Handlebars runtime library. Any other templates we were using would also be available via this object.

- Given a photo URL and some caption text, we could call our photo card template function to produce a corresponding HTML photo card. Our arguments need to be passed into the template function within an object. The keys of this object must match the identifiers within our template, and the corresponding values in the object must be the content we want to insert at those identifiers.
    - This object containing the template arguments is specifically called the ***context***. We'll see more powerful ways the context can be used later on.

- For example, if we wanted to create a photo card containing a cute kitten photo, we could do something like this:

```
var photoCardContext = {
  photoURL: "http://placekitten.com/320/320/",
  caption: "A very cute kitty."
};

var photoCardHTML =
  Handlebars.templates.photoCard(photoCardContext);
```

- Now, the variable `photoCardHTML` will contain a string of HTML content. Specifically, it will contain this HTML string representing our new kitty photo card:

```
<section class="photo-card">
  <div class="img-container">
    <img class="person-photo-img"
      src="http://placekitten.com/320/320/" />
  </div>
```

```
  <div class="caption">
    A very cute kitty.
  </div>
</section>
```

- At this point, our new photo card isn't inserted into the page. It only exists as an HTML string. We can use the `insertAdjacentHTML()` method to actually place that HTML into the DOM. Specifically, we can grab the container element into which we want to insert the new photo card using `getElementById()`, `querySelector()`, etc. and then call `insertAdjacentHTML()` on that element:

```
var photoCardContainer = document.getElementById(...);
photoCardContainer.insertAdjacentHTML('beforeend',
  photoCardHTML);
```

- Here, the argument `'beforeend'` indicates that we want to place the new photo card within the container element *after* all of the other contents of that element. We could also use `'afterbegin'` to insert the new photo card within the container element *before* its other contents.
    - `'beforebegin'` and `'afterend'` are also possible arguments. You can refer to the docs for `insertAdjacentHTML()` for more info.

Handlebars views and partials

- Using templates to render new components on the client-side is pretty great, especially if the components we want to render have even slightly complex structure, since it's much easier to write a Handlebars template to represent HTML structure than it is to use native JS methods to do so.

- However, pages still have to be rendered on the server side as well as on the client side in order to satisfy an initial request for a page.

- What we'd like to be able to do is to use our templates on the server side, as well, so that we can write a single representation (i.e. a template) of each component that will be re-used/repeated across our site. Luckily, we can do this easily. We'll look specifically at how to do it using Handlebars templates in an Express-based server.

- Specifically, we'll see how to write Handlebars ***views***. A view is a specific term that is used to describe a template representing an entire page.

- The first thing we'll do is write a view template to represent an entire page of photo cards. Within this view template, we'll invoke the photo card template we wrote above each time we want to render a single photo card.

- We'll start by writing the outer HTML structure of the page and leaving space where we'll eventually place the photo cards. The outer HTML structure of the page might look something like this:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Photos</title>
    <link rel="stylesheet" href="/style.css">
    <script src=".../handlebars.runtime.js" defer></script>
    <script src="/photoCardTemplate.js" defer></script>
    <script src="/index.js" defer></script>
  </head>
  <body>
    <main id="photo-card-container">
      <!-- Photo cards go here. -->
    </main>
  </body>
</html>
```

- Note that at this point, we haven't included any template expressions yet. Note also that we're not including any photo cards yet. Right now, we just have a view representing an empty page.

- Like we said above, to include photo cards, we want to invoke the photo card template we wrote above within our larger template. Handlebars allows us to do

this using ***partials***, which are simply templates we invoke within other templates to render components of our site.

- In Handlebars, we invoke a partial with syntax like this: `{{> partialName}}`. For example, if we wanted to invoke our photo card template from within another template (such as our photos page view), we could do so like this:

```
{{> photoCard}}
```

- Here, the identifier `photoCard` is derived from the filename in which we wrote our photo card template: `photoCard.handlebars`.

- Note that the photo card partial needs to be passed a context containing the properties `photoURL` and `caption` in order to correctly render a complete photo card. The invocation above of the photo card template as a partial does not contain any specification of the context.
- In Handlebars, if we don't specify any additional arguments in the invocation of our photo card partial (or any partial), the current context in the template from which we're invoking the partial would be passed as the partial's context.

- For example, if the current context in the template from which we're invoking our photo card partial already has the properties `photoURL` and `caption` and we invoke the photo card partial as above without arguments, then the values of those properties will be used in the corresponding Handlebars expressions in our photo card partial.

- Alternatively, we could specifically pass an object into our partial, and this object would become the context for the partial. For example, if within the current context in the template from which we're invoking our photo card partial we had an object named `photoInfo` with the properties `photoURL` and `caption`, we could specifically pass that object into our partial invocation to serve as the partial's context, like this:

```
{{> photoCard photoInfo}}
```

- If we wanted to, we could even pass values for specific properties into the partial like this:

```
{{> photoCard photoURL="..." caption="..."}}
```

Built-in Handlebars helpers

- Now that we know how to pass a context into our photo card partial, we need to be able use that partial to render *multiple* photo cards. We could do this by simply placing multiple invocations of the partial into our photos page view, but that would limit us to always render the specified number of photo cards. For some pages, we may want to render more or fewer photo cards than the number of times we chose to invoke the photo card partial.

- We can accomplish this by passing an *array* of objects into our photos page view, where each of those objects represents one photo with the properties `photoURL` and `caption`.

- Let's say we name this array `photos` within the context we pass to the photos page view. Then, we can render one photo card for each object in that array using Handlebars' `each` helper:

```
{{#each photos}}
  {{> photoCard}}
{{/each}}
```

- The `each` helper is a piece of functionality built in to Handlebars that simply loops through each element in an array (or each property in an object) specified as an argument to the helper and repeats the content specified within the `{{#each}}{{/each}}` block once for each of those elements.

- Thus, the `each` block above simply invokes the photo card partial once for each photo in the array `photos`. If we put such a loop in our photos page template where we want to render the photo cards, we only need to make sure to pass an appropriately-structured array called `photos` in the context we pass to the photos page view.

- Importantly, note that we are not specifically passing any context into the photo card partial when we invoke it within the `each` helper block above. This is because the `each` helper iterates through the elements of the specified array,

and within the `{{#each}}{{/each}}` block, the context is automatically set to the current element in the iteration.

- In other words, say our `photos` array contains several objects of the following form, each representing an individual photo:

```
{
  photoURL: "...",
  caption: "..."
}
```

  Then in each iteration of the `each` helper, the context will automatically be set to be one of those objects.

- Since the current context is passed as the context into a partial if we don't specify any arguments when invoking that partial, this means that each iteration of the `each` helper will result in one of our photo-representing objects being passed as the context into the photo card view.

- The result will be that one photo card will be rendered for each photo object in our `photos` array.

- There are other helpers built in to Handlebars in addition to `each`. The `if` helper is also particularly useful. The `if` helper evaluates a value in the current template context as a boolean. If that value evaluates to `true`, then the content within the `if` helper block is displayed. Otherwise it isn't.

- For example, in the snippet below, the `<p>` element will be displayed if the current template context contains a value `displayTheP` that evaluates to `true`. If the current template context either does not contain a value `displayTheP`, or if that value evaluates to `false`, then the `<p>` element will not be displayed:

```
{{#if displayTheP}}
  <p>This paragraph will be rendered, or it might not.</p>
{{/if}}
```

- An `{{else}}` block can also be included when using the `if` helper, e.g.:

```
{{#if displayTheP}}
```

```
  <p>This paragraph might be rendered, or it might not.</p>
{{else}}
  <p>This will be rendered if the paragraph above
isn't.</p>
{{/if}}
```

- There is also a built-in `unless` helper which works much like the `if` helper but displays its content if its argument evaluates to `false`:

```
{{#unless doNotDisplayTheP}}
  <p>This is rendered if doNotDisplayTheP is falsy.</p>
{{/unless}}
```

- Like with the `if` helper, an `{{else}}` block can also be used with the `unless` helper.

- There are other built-in helpers available in Handlebars, too.

Rendering view templates from an Express server

- Let's put all these pieces together and start using our templates to render pages dynamically on the server.

- The first thing we'll need to do is install a server-side dependency named `express-handlebars`, which will allow us to incorporate Handlebars into an Express-based server:

```
npm install --save express-handlebars
```

- In Express, there is a response method called `res.render()` that can be called to use a view template to dynamically render a page and send it back as a response to the client. Once `express-handlebars` is installed, we can hook our Express server up so that calls to `res.render()` uses Handlebars and our templates to perform that rendering.

- First, we need to make sure our templates live in the places Express and `express-handlebars` expect to find them.

- Templates for views need to live in a server package directory named `views/`, so we can save the photos page view template we wrote above in a file called `views/photosPage.handlebars`.

- Partials need to live in a server package directory named `views/partials/`, so we can save our photo card partial template in a file called `views/partials/photoCard.handlebars`.
    - Note that we'll have to make sure the build script we hooked up above to compile our photo card template and make it available for use on the client side knows where to find the photo card template.

- Once all of our templates are stored in the correct directories, we need to tell our Express server to use `express-handlebars` to render content whenever `res.render()` is called.
- To do this, we'll first simply import our `express-handlebars` dependency in our server script:

```
var exphbs = require('express-handlebars');
```

- Then, once we have our express server `app` created, we need to first register `express-handlebars` with Express as a template engine, and then we need to tell Express to *use* `express-handlebars` by default every time `res.render()` is called:

```
app.engine('handlebars', exphbs());
app.set('view engine', 'handlebars');
```

    - In these two lines, the string `'handlebars'` is assigned and used as an identifier for `express-handlebars`.

- With those pieces in place, we're ready to call `res.render()` to render content with our templates.

- Remember, when we use a Handlebars template to render content, we need to pass a context object into the template that contains the dynamic data the template will use to fill out the template. The property names in our context object need to match the identifiers we used in the expressions in our templates.

- In this case, we wrote our photos page view template to expect a context containing a property named `photos` whose value is an array, each element of which is an object representing a single photo with the properties `photoURL` and `caption`. When we call `res.render()`, we'll need to pass such a context as an argument.

- For now, let's assume we have an array of photo objects living in our server's memory, e.g.:

```
var photoObjects = [
  {
    photoURL: "...",
    caption: "...",
  },
  ...
];
```

- Then, our call to `res.render()` within the route in which we want to render our photos page might look like this, where the string `'photosPage'` is derived from the name the file where we stored our photos page template, `photosPage.handlebars`:

```
res.render('photosPage', {
  photos: photoObjects
});
```

    - Here, the second argument is the context object that's being passed into our photos page view. If we needed to, we could add other properties to this object to provide additional arguments to our template.

- This call will result in our server process generating HTML representing a photos page that contains one photo card for each photo in the array `photoObjects`.

- We could do something even more interesting if we had many arrays of photo objects, e.g. one photo array for each of several people. We could store these in an object whose keys served as identifiers of the people pictured in the corresponding array of photo objects (each of which had the properties `photoURL` and `caption`):

```
var photoData = {
  luke: [ /* Array of photo objects for Luke */ ],
  leia: [ /* Array of photo objects for Leia */ ],
  han: [ /* Array of photo objects for Han */ ],
  r2d2: [ /* Array of photo objects for R2-D2 */ ],
  ...
};
```

- Such an object looks very much like what we might store in a database powering our application. For now, though, we can assume we have this object hard-coded somewhere.

- With such an object in place, we can write a route that dynamically renders a photos page for whatever person the client requests, where the requested person is identified as a parameter in the request path:

```
app.get('/photos/:personId', function (req, res, next()) {
  var personId = req.params.personId;
  if (photoData[personId]) {
    res.render('photosPage', {
      photos: photoData[personId]
    });
  } else {
    next();
  }
});
```

- This route will render a page of Luke's photos whenever a client requests the path `/photos/luke`. Or, if the client requests, `/photos/leia`, it will render a page of Leia's photos. It will do something similar for `/photos/han`, `/photos/r2d2`, etc. If the person specified in the request path doesn't match anyone in our `photoData` object, the route will simply call the next middleware function, implying that there is no photos page for the requested person.

Using Handlebars layouts

- We've done a great job so far at eliminating duplicated code in our project. Now, every time a photo card is rendered, whether it's on the server or on the client, it will be rendered using our photo card template, which is the only representation of the photo card.

- Similarly, whenever a photos page is rendered, no matter for what person, the same photos page view template will be used, and this is our only representation of the photos page.

- What happens, though, if our site has many pages, and these pages all have similar structure? For example, what if our site has a home page, a page listing out all of the people for whom we have photos, a 404 page, etc. All of these pages might look very much the same, and they might use the exact same HTML skeleton.

- Ideally, we'd like to be able to abstract this common *outer* structure away into its own template, so we can implement it once and then reuse that implementation.

- Again, this is something `express-handlebars` allows us to do using ***layout templates***. A layout template provides an HTML "wrapper" for templates representing pages, and typically includes a specification of the HTML skeleton common to all pages in a site. A layout can go further, though, containing visual components common to all pages (e.g. header and navbar).

- We can write a simple layout template for our photos site. To do so, we can create a new file called `views/layouts/main.handlebars` with the following contents:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Photos</title>
    <link rel="stylesheet" href="/style.css">
    <script src=".../handlebars.runtime.js" defer></script>
```

```
    <script src="/photoCardTemplate.js" defer></script>
    <script src="/index.js" defer></script>
  </head>
  <body>
    {{{body}}}
  </body>
</html>
```

- This layout contains the exact outer HTML skeleton from our photos page template. However, we're replaced the main content of the photos page with a Handlebars expression `{{{body}}}`. This provides a placeholder into which the main content of our site's pages should be rendered.
    - Importantly, note that the `{{{body}}}` expression is enclosed in *3* sets of curly brackets. In Handlebars, content placed into such expressions *is not escaped*, meaning that HTML placed into a such an expression *will be rendered as HTML*.

- With this template in place, we can modify our photos page view template in `views/photosPage.handlebars` to contain *only* the content specific to the photos page:

```
<main id="photo-card-container">
  {{#each photos}}
    {{> photoCard}}
  {{/each}}
</main>
```

- Finally, we need to tell `express-handlebars` to use our layout when rendering a page. We can do this when we register `express-handlebars` with Express as a view engine, modifying the following line to add an arguments object to the call to `exphbs()`:

```
app.engine('handlebars', exphbs({defaultLayout: 'main'}));
```

    - Here, the string `'main'` is derived from the name of our layout template file, `views/layouts/main.handlebars`.

- Now, every template we render by calling `res.render()` will be plugged into the `{{{body}}}` placeholder in our layout template. For example, we could

write a very simple view template for our home page in `views/homePage.handlebars`:

```
<main>
  <h1>Welcome to our photos site!</h1>
</main>
```

- This could be rendered in a simple route like this:

```
app.get('/', function (req, res) {
  res.render('homePage');
});
```

  - Note that here the home page template doesn't expect any arguments, so we don't pass a context to it in our call to `res.render()`.

- A 404 page or a page listing the people for whom we had photos pages would be similarly simple, and at this point, we've seen how to factor everything so that there would need to be essentially no code duplicated between our representations of each of those pages and their components.

- To summarize, partials are templates representing components of our site. They can be invoked within other partials or within views, which are templates representing whole pages in our site. Finally, a layout is a template that represents an outer "wrapper" for all of the pages in our site. After a view is rendered into HTML, that HTML is placed into the appropriate location in a layout. This relationship looks like this: