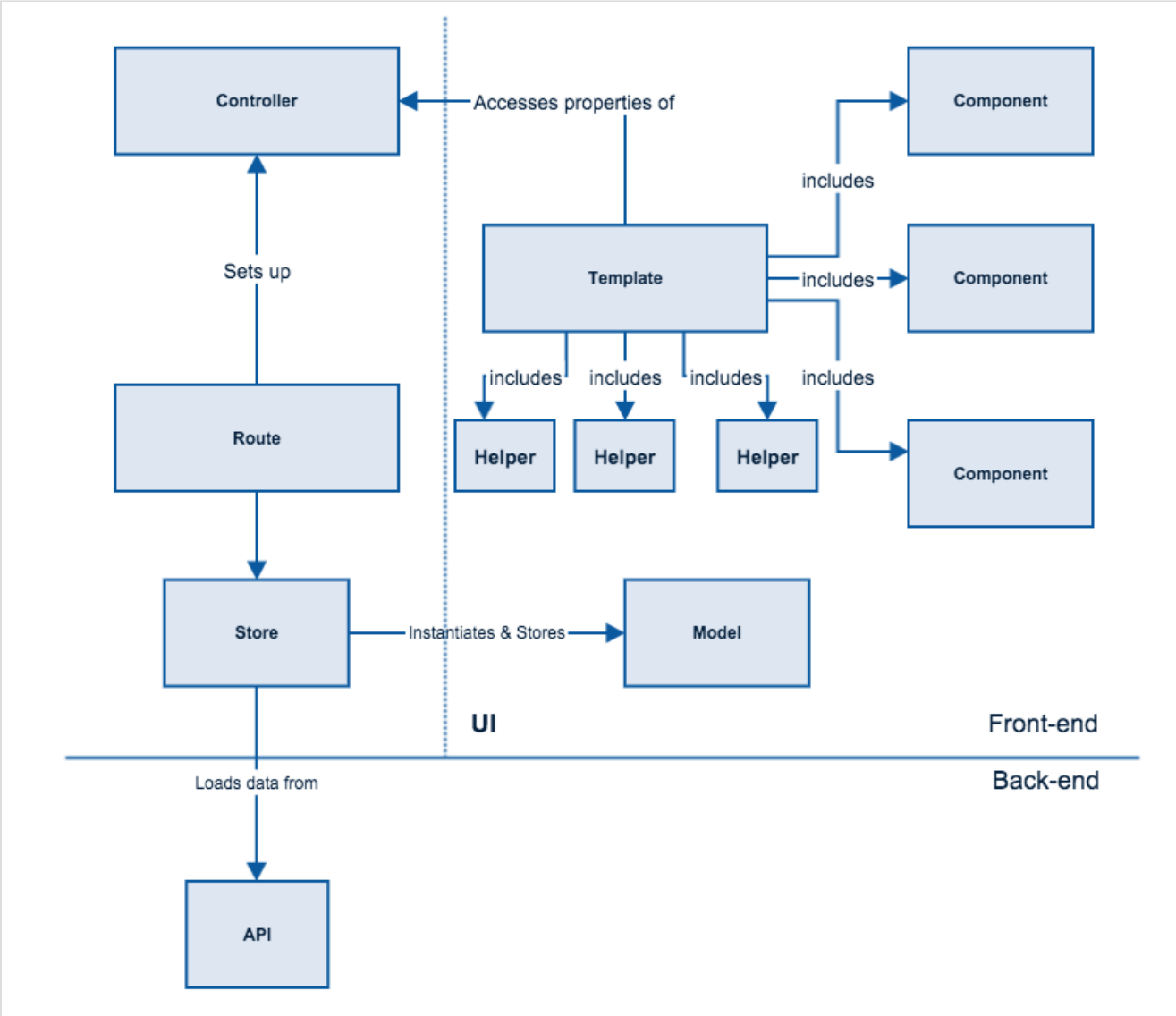
Ember Building Blocks

This document shows how components of an Ember application work together, what the responsibility of each of them is and how they can access other parts of the application.

Because components are themselves an important component of Ember applications in the above sense, I'm instead going to use the term "Ember Building Blocks" (EBBs) to refer to routes, models, controllers, components, etc. throughout the document.

The following graph gives a very high-level overview of how EBBs cooperate to make your applications work.



Let's now see what each piece does and how it can reach the other pieces (those that it needs to).

Routes

Since routes orchestrate the flow of data in Ember applications and set up the rendering context for the templates, they should have access to their controller and other routes. Let's see how.

Model of the route

- `this.modelFor(this.routeName)` – Be wary that the model is only set at the end of the `model` hook so only use it with a guard in the `beforeModel` and `model` hooks.

- `this.controller.get('model')` - `this.controller` might be undefined, see comment in the "Controller of the route" point

Models set in other routes

- `this.modelFor(someRouteName)` - this will only work for routes whose model has already been fetched, in other words, the parent routes of the route this is called from.

Store

- `this.store` - In non-trivial applications, the route usually communicates with the back-end API through the store service. This is provided both by Ember Data and ember-model or can be implemented by the application developer. The store service instantiates the model objects of the application (see the section on "Models") either from the server response or explicitly in the Ember app. The store acts as a caching layer between the Ember app and the back-end so that loading a certain instance that had already been fetched need not hit the API.

Controller of the route

- `this.controller` - Be aware that the controller for a route is only created in the `setupController` step of the route transition so if you use it in any of the model hooks, you should guard against the controller not being there yet.

Other controllers

- `this.controllerFor(someRouteName)` - returns the controller belonging to a parent route (see the "Models set in other routes" section). I'm wary of accessing other controllers and so I recommend against using this method.

Models

- Models represent the object types used in your application and encapsulate the properties. Most of them have a 1:1 mapping to the domain objects of your application, the "business logic".

- Since they are about defining model data, composing that model data to a new form (e.g `fullName`) or computing new properties from properties (`isMajor`), they should not have to access other EBBs of the application.

Controllers

The model belonging to the corresponding route

- `this.get('model')` - The `model` property of the controller is set to the value returned by the corresponding route's model hook. If a promise is returned, it is resolved before the controller is instantiated.

```js
// app/routes/bands.js
import Ember from 'ember';

const { Route } = Ember;

export default Route.extend({
  model() {
    return this.store.findAll('band');
  }
});
```

```js
// app/controllers/controller.js
import Ember from 'ember';

const { Controller } = Ember;

export default Controller.extend({
  bandCount() {
    /*
    this.get('model') is the collection of bands
    returned by the above model hook, so its length
    is the number of bands returned.
    */
    return this.get('model.length');
  }
});
```

There is no use to guard against `this.get('model')` being `undefined` here as the `model` property of the controller is set in the route. It can still be `undefined` if that's what the `model` hook returned but then it's probably unwanted behavior and should throw an error.

Components

- Components implement a piece of UI and should be isolated from their surroundings. They should only be able to access things that are passed to them as properties.

- To communicate with the outside world, they use actions that are, again, passed to them as properties.

```hbs
{{star-rating item=song rating=song.rating on-click=(action "updateRating")}}
```

The component can then access the `item`, `rating`, and `on-click` properties:

```
import Ember from 'ember';

const { Component } = Ember;

export default Component.extend({
  classNames: ['rating-panel'],

  rating:     0,
  item:       null,
  "on-click": null,
  (...)
});
```

Given that the main purpose of components is isolation and reusability, you shouldn't access any other Ember building blocks from inside components. The one exception might be injecting a service into the component.

Think about an activity feed component that polls an API endpoint to see if there are any new notifications and thus needs the ajax service:

```
// app/components/activity-feed
import Ember from 'ember';

const { inject } = Ember;

export default Ember.Component.extend({
  ajax: inject.service(),

  poll: function() {
    this.get('ajax').request('/notifications', function() {
      (...)
    });
  }
});
```

Component lifecycle hooks

These hooks are called at different points of the component's lifecycle. Some are only called once, on initial render, others are called at every re-render.

If you've heard that observers are "bad", but wasn't sure how to implement features without them in components, lifecycle hooks are probably the answer.

I'm only going to list the most frequently used ones. You can find more information about each in the official guides.

For an example, let's use the following component invocation:

```
{{star-rating item=song rating=song.rating on-click=(action "updateRating")}}
```

The attributes of this component are the keys in the key value pairs, so in this case, `item`, `rating` and `on-click`. I like to think about component invocations as function calls. In this analogy, we'd call the component with named arguments, and the function argument names are the attributes of the component.

Now, back to the lifecycle hooks:

- `init`

This is not strictly a component lifecycle event as all Ember objects have it. It gets called when the object is created and gives you a chance of initializing the object's properties.

- `didReceiveAttrs`

This hook gets called both when the component invocation is originally made (called "initial render") and when an attribute has changed (or `this.rerender` is explicitly called) and thus the component is re-rendered. A fitting use-case would be to transform values passed in to the internal format the component uses. (converting strings to numbers, for example.)

- `didInsertElement`

This gets called once in the component's lifecycle, when its HTML representation has been inserted into the DOM. This is classically used to integrate with 3rd party libraries (like jQuery plugins, if you have to use them) but is also suitable to be used for setting up things that should live while the component lives, like timers.

- `didUpdateAttrs`

This is only called at re-render time, when some attribute of the component has changed. You could, for example, use this to keep track of whether a certain attribute (the `rating` in the above invocation) has changed and then trigger an action based on that change (animate to the new number of stars, for example).

- willDestroyElement

The other half of `didInsertElement`, this hook gets called when the component is about to be destroyed, giving us a chance to tear down what'd been set up in `didInsertElement` (for example, stopping polling).

Route-driven templates

- The controller belonging to the same route serves as the context of the route-driven template. Consequently, any property in the template is looked up on the controller:

```JS
// app/controllers/bands.js
import Ember from 'ember';

const { Controller } = Ember;

export default Controller.extend({
  bandCount() {
    return this.get('model.length');
  }
});
```

```HBS
// app/templates/bands.hbs
There are {{bandCount}} bands.
```

- Currently, helpers and components, even those in installed add-ons, are all registered in a global scope and are accessible from any template:

```HBS
// app/templates/bands/band/songs.hbs
<ul class="list-group songs">
  (...)
  {{#each sortedSongs as |song|}}
    <li class="list-group-item song">
      {{capitalize song.title}}
      {{star-rating item=song rating=song.rating on-click=(action
"updateRating")}}
    </li>
  {{/each}}
</ul>
```

Here, `capitalize` is a helper while `star-rating` is a component. Currently there is no way of telling them apart (other than components must have a dash in their name) as they both live in the same global space. Furthermore, what happens if you have both a component and a helper with the same name is undetermenistic, so you should strive to avoid this scenario.

The Module Unification RFC addresses this problem by allowing local lookups, so if you want to learn more about it, head there.

- To maintain a clear separation of concerns and have a readable and maintainable code base, you cannot reach any other Ember building blocks from your templates, which is a very good thing.

Component templates

- You can think of component templates as the component's html output.
- The context of component templates is its component, so properties defined on the component can be accessed from the template.
- Using the above `star-rating` component example, if the component has a `stars` property defined on it, you can write the component's template like this:

```hbs
{{#each stars as |star|}}
  <a href="#" {{action "setRating" star.rating}}
     class="star-rating {{if star.full fullClassNames emptyClassNames}}">
  </a>
{{/each}}
```

Helpers

Helpers are simple functions that you can use to transform some input data for output, such as numbers (e.g format-currency), dates (e.g date-from-now) and text (e.g capitalize).

They only have access to their parameters.

```js
// app/helpers/round.js
import Ember from 'ember';
const { Helper } = Ember;

export function round(number) {
  return Math.round(number[0]);
}

export default Helper.helper(round);
```

(The example is taken from ember-math-helpers.)

Services

- You can get access to a service by injecting it into the class you want to use it in and then accessing via a standard `get`:

```js
// app/services/user-session.js
import Ember from 'ember';

const { Service } = Ember;

export default Service.extend({
  (...)
});
```

```js
// app/routes/bands.js
import Ember from 'ember';

const { Route, inject } = Ember;

export default Route.extend({
  session: inject.service('userSession'),

  model() {
    if (this.get('session.isAuthenticated')) {
      (...)
    }
  }
});
```

This is true for routes, controllers, components and any other object that was initialized through the container.