# Title: Front End Best Practices

## General

### Use utils over mixins

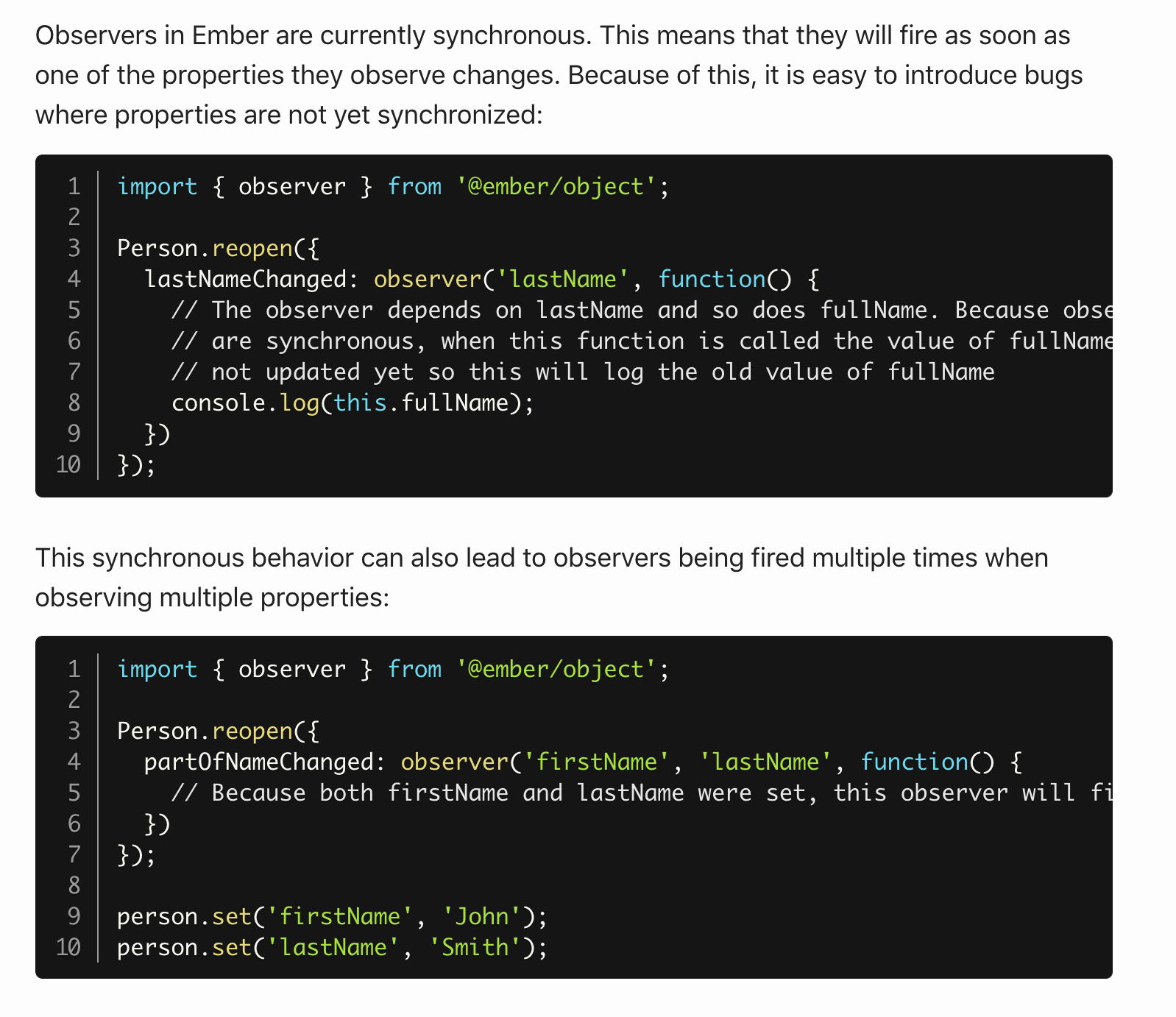
We are using mixins at a lot of places, the reason we are switching to utils is because its hard to identify what all functions of the mixin that we are using and we are not able to spot out if any 2 mixins have same method names that is being overridden  
  
For example:

* lets say we have 5 mixins and we have to find out where a particular function sampleFunction() is being consumed from, you would need to go into each mixin to check if it's there or not  
  more over there are cases where a mixin might be using another mixin
* When 2 mixins have methods with same name, one of them gets overridden, resulting in side effects.

Refer: utils/get-filter-views for an example

Don't use Observers

Ember.Observer should not be used. alternatives are component life cycle events and actions



We may have wrap the code around with run.once method to overcome these shortages and thats additional work which could probably introduce bugs if the developer forgets to do so.

### Don't use Select-2

Select-2 package is deprecated on 2015, we are migration to power-select. Please refrain from introducing new Select-2 and awesome if you migrated select-2 in your working code to power-select

### Don't use this.attrs

Don't use this.attrs when accessing properties passed to a component, you can directly access this.firstName instead of this.attrs.firstName inside component where firstName is the property passed into the component. [Read more](https://locks.svbtle.com/to-attrs-or-not-to-attrs)

Example

{{example-component firstName=firstName editorReference=(action editorReference)}}  
  
// inside example-component  
  
// Good  
get(this,'firstName');  
  
this.editorReference();  
  
// Bad  
get(this.attrs, 'firstName'); or get(this, 'attrs.firstName');

this.attrs.editorReference();

### Dont use the arguments passed to the didInitAttrs, didReceiveAttrs and didUpdateAttrs

These arguments are private API, imposes an unnecessary performance hit on all components whether they are used or not, and can be easily replicated by the users in cases where they are needed. Refer <https://github.com/emberjs/rfcs/blob/master/text/0191-deprecate-component-lifecycle-hook-args.md>

### Use get and set

Calling someObj.get('prop') couples your code to the fact that someObj is an Ember Object. It prevents you from passing in a POJO, which is sometimes preferable in testing. It also yields a more informative error when called with nullor undefined.

Although when defining a method in a controller, component, etc. you can be fairly sure this is an Ember Object, for consistency with the above, we still use get/set.

Example

// Good  
import { get, set } from '@ember/object';  
  
set(this, 'isSelected', true);  
get(this, 'isSelected');  
  
// Bad  
  
this.set('isSelected', true);  
this.get('isSelected');

### Override init

Rather than using the object's init hook via on, override init and call \_super with ...arguments. This allows you to control execution order.

### Alias your model

It provides a cleaner code to name your model user if it is a user. It is more maintainable, and will fall in line with future routable components

Example

export default Controller.extend({  
 user: alias('model')  
});

### Use Ember Concurrency

When ever you make an async request make sure you always do via ember concurrency except if it is in the model hook of the route. [Read More](https://youtu.be/VEzVDOmY-dc)

### Avoid the usage of computed.oneWay and computed.reads

With oneWay and reads, properties act as an alias to their dependent property *until they are set*. This means that the two properties, once representing the same data, *diverge* and no longer represent the same value. Suddenly, the data flow has been broken. The original property and oneWay property no longer have a connection, as it's understood that the oneWay property has been updated in a way that isn't expected to affect the original object. You can use readOnly as an alternative. [Read more](https://m.alphasights.com/the-risks-you-take-when-you-use-ember-computed-oneway-f2c008a14404)

### Avoid using multiple set

multiple set causes hinderance in Ember.runway pipeline so it is recommended to call setProperties together

### Avoid using multiple Ember.on

The use of on(), which is one of the patterns, is **STRONGLY** discouraged. This primarily has to do with the execution order of the functions attached to these events. on() is scheduled in the events queue, so if you define more than one on() against an event you will not be able to determine the execution order of the on()s in relation to one another. This indeterminate execution order also presents itself when mixins are employed or inheritance is changed. Here is the [link](https://notmessenger.com/proper-use-of-ember-on/) to read more

### Avoid using sendAction

sendAction is deprecated and is to be removed going forward you can use Ember closure actions

### Don't use stateManager

This was mainly invented for the need of transitioning from a component to another route. Which is no longer needed as we can use [Embers router service](https://github.com/rwjblue/ember-router-service-polyfill) itself

### Inject services as required

You don't need all services in all components, routes, controllers and so on, as of now we have built it such a way that it is injected everywhere but moving forward we want to separate this as we don't want the dependency injection to be present in all modules unnecessarily.

### Use proper variable and function names

The name of a variable, function, or class, should answer all the big questions. It should tell you why it exists, what it does, and how it is used. If a name requires a comment, then the name does not reveal its intent. In short names should be:

1. intention revealing
2. pronounceable
3. searchable

##### Class names

Classes and objects should have noun or noun phrase names like Customer, WikiPage, Account, and AddressParser. Avoid words like Manager, Processor, Data, or Info in the name of a class. A class name should not be a verb.

##### Method Names

Methods should have verb or verb phrase names like postPayment, deletePage, or save.

Example

// Good  
let firstName;  
let lastName;  
let counter;

const MAX\_CART\_SIZE = 100;   
const isFull = cart.size > MAX\_CART\_SIZE

// Bad  
let fName;  
let lName;  
let cntr;  
let full = false;   
  
if (cart.size > 100) {   
 full = true   
}

### Write Modular and Specialised functions

Your functions should do one thing only on one level of abstraction.

Example

// DON'T

function getUserRouteHandler (req, res) {  
  const { userId } = req.params  
  // inline SQL query  
  knex('user')  
  .where({ id: userId })  
  .first()  
  .then((user) => res.json(user))  
}

// DO  
// User model (eg. models/user.js)

const tableName = 'user'  
const User = {  
  getOne (userId) {  
  return knex(tableName)  
  .where({ id: userId })  
  .first()  
  }  
}

// route handler (eg. server/routes/user/get.js)

function getUserRouteHandler (req, res) {  
  const { userId } = req.params  
  User.getOne(userId)  
  .then((user) => res.json(user))  
}

### Avoid long argument list

Use a single object parameter and destructuring assignment instead. It also makes handling optional parameters much easier.

Example

// DON'T  
function getRegisteredUsers (fields, include, fromDate, toDate) { /\* implementation \*/ }

getRegisteredUsers(['firstName', 'lastName', 'email'], ['invitedUsers'], '2016-09-26', '2016-12-13')  
  
// DO  
function getRegisteredUsers ({ fields, include, fromDate, toDate }) { /\* implementation \*/ }  
  
getRegisteredUsers({  
fields: ['firstName', 'lastName', 'email'],  
include: ['invitedUsers'],  
fromDate: '2016-09-26',  
toDate: '2016-12-13'  
});

### Use pure functions without side effects

Use pure functions without side effects, whenever you can. They are really easy to use and test.

### Don't use JS for styling

JavaScript is good for calculation, conversion, access to outside sources (Ajax) and to define the behavior of an interface (event handling). Anything else should be kept to the technology we have to do that job.

Example

// DON'T  
function addItemToCart (cart, item, quantity = 1) {  
  const alreadyInCart = cart.get([item.id](http://item.id)) || 0;  
  cart.set([item.id](http://item.id), alreadyInCart + quantity);  
  return cart;  
}  
  
  
  
// DO  
// not modifying the original cart  
function addItemToCart (cart, item, quantity = 1) {  
  const cartCopy = new Map(cart);  
  const alreadyInCart = cartCopy.get([item.id](http://item.id)) || 0;  
  cartCopy.set([item.id](http://item.id), alreadyInCart + quantity);  
  return cartCopy;  
}  
  
// or by invert the method location  
// you can expect that the original object will be mutated  
// addItemToCart(cart, item, quantity) -> cart.addItem(item, quantity)  
const cart = new Map();  
Object.assign(cart, {  
  addItem (item, quantity = 1) {  
  const alreadyInCart = this.get([item.id](http://item.id)) || 0;  
  this.set([item.id](http://item.id), alreadyInCart + quantity);  
  return this;  
  }  
});

### Allow for Configuration

Everything that is likely to change in your code should not be scattered throughout your code. This includes labels, CSS classes, IDs and presets.

### Avoid Heavy Nesting

A really bad idea is to nest loops inside loops as that also means taking care of several iterator variables (i,j,k,l,m...).

Example

// DON'T  
function renderProfiles(o){  
  var out = document.getElementById('profiles');  
  for(var i=0;i<o.members.length;i++){  
     var ul = document.createElement('ul');  
     var li = document.createElement('li');  
     li.appendChild(document.createTextNode(o.members[i].name));  
     var nestedul = document.createElement('ul');  
     for(var j=0;j<o.members[i].data.length;j++){  
        var datali = document.createElement('li');  
        datali.appendChild(  
           document.createTextNode(  
              o.members[i].data[j].label + ' ' +  
              o.members[i].data[j].value  
           )  
        );  
        nestedul.appendChild(detali);  
     }  
     li.appendChild(nestedul);  
  }  
  out.appendChild(ul);  
}  
  
  
  
// DO  
function renderProfiles(o){  
  var out = document.getElementById('profiles');  
  for(var i=0;i<o.members.length;i++){  
     var ul = document.createElement('ul');  
     var li = document.createElement('li');  
     li.appendChild(document.createTextNode(data.members[i].name));  
     li.appendChild(addMemberData(o.members[i]));  
  }  
  out.appendChild(ul);  
}  
function addMemberData(member){  
  var ul = document.createElement('ul');  
  for(var i=0;i<member.data.length;i++){  
     var li = document.createElement('li');  
     li.appendChild(  
        document.createTextNode(  
           member.data[i].label + ' ' +  
           member.data[i].value  
        )  
     );  
  }  
  ul.appendChild(li);  
  return ul;  
}

### Use === Instead of ==

JavaScript utilizes two different kinds of equality operators: === | !== and == | != It is considered best practice to always use the former set when comparing.

### Comment Your Code

Comment your code as best as possible. It helps when returning to the implementations later to help recollect the line of thinking was. Also when colleagues needs to revise your code? Always, always comment important sections of your code.

Organising modules

Ordering a module's properties in a predictable manner will make it easier to scan.

###### Services

* Inject services first. Eg: session, i18n, etc.

###### Plain properties

* They are properties that configure the module's behavior. Examples are tagName and classNames on components and queryParams on controllers and routes. Followed by any other simple properties, like default values for properties.

###### Single line computed property macros

* E.g. alias, sort and other macros. Start with service injections. If the module is a model, then attr properties should be first, followed by belongsTo and hasMany.

###### Multi line computed property functions



###### Lifecycle hooks

* The hooks should be chronologically ordered by the order they are invoked in.

###### Functions

* Public functions first, internal functions after.

###### Actions

Example

export default Component.extend({  
 // Plain properties  
 tagName: 'span',  
  
 // Single line CP  
 post: alias('myPost'),  
  
 // Multiline CP  
 authorName: computed('author.{firstName,lastName}', function() {  
 // code  
 }),  
  
 // Lifecycle hooks  
 didReceiveAttrs() {  
 this.\_super(...arguments);  
 // code  
 },  
  
 // Functions  
 someFunction() {  
 // code  
 },  
  
 actions: {  
 someAction() {  
 // Code  
 }  
 }  
});

## Templates

### Do not use partials

Always use components. Partials share scope with the parent view, use components will provide a consistent scope.

### Don't yield this

Use the hash helper to yield what you need instead.

Example

{{! Good }}  
{{yield (hash thing=thing action=(action "action"))}}  
  
{{! Bad }}  
{{yield this}}

### Use components in {{#each}} blocks

Contents of your each blocks should be a single line, use components when more than one line is needed. This will allow you to test the contents in isolation via unit tests, as your loop will likely contain more complex logic in this case.

Example

{{! Good }}  
{{#each posts as |post|}}  
 {{post-summary post=post}}  
{{/each}}  
  
{{! Bad }}  
{{#each posts as |post|}}  
 <article>  
 <img src={{post.image}} />  
 <h1>{{post.title}}</h2>  
 <p>{{post.summar}}</p>  
 </article>  
{{/each}}

### Always use the action keyword to pass actions

Although it's not strictly needed to use the action keyword to pass on actions that have already been passed with the action keyword once, it's recommended to always use the action keyword when passing an action to another component. This will prevent some potential bugs that can happen and also make it more clear that you are passing an action.

Example

{{! Good }}  
{{edit-post post=post deletePost=(action deletePost)}}  
  
{{! Bad }}  
{{edit-post post=post deletePost=deletePost}}

### Ordering static attributes, dynamic attributes, and action helpers for HTML elements

Ultimately, we should make it easier for other developers to read templates. Ordering attributes and then action helpers will provide clarity.

Example

{{! Bad }}  
  
<button disabled={{isDisabled}} data-auto-id="click-me" {{action (action click)}} name="wonderful-button" class="wonderful-button">Click me</button>

{{! Good }}  
  
<button class="wonderful-button"  
 data-auto-id="click-me"  
 name="wonderful-button"  
 disabled={{isDisabled}}  
 onclick={{action click}}>  
 Click me  
</button>

**Use SVG instead of Icon Fonts**

We are migrating from icon fonts to svg  
  
 Read more:

* [Why GitHub switched from an icon font to SVG](https://github.com/blog/2112-delivering-octicons-with-svg)
* ["Inline SVG vs icon fonts" from css-tricks](https://css-tricks.com/icon-fonts-vs-svg/)
* [Ten reasons to switch from an icon font to SVG](http://ianfeather.co.uk/ten-reasons-we-switched-from-an-icon-font-to-svg/)

Example

// good  
{{svg-jar "edit" class="svg-sm"}}  
  
// bad  
<span class="icon-edit"></span>

### Prefer computed properties instead of heavy conditioning in Templates

Reducing the logic in templates helps in increasing the performance in the later versions of ember.

//Not recommended  
{{if (or (and didCustomise isBulk) isDeal)}}  
 <button>Customise</button>  
{{/if}}  
  
  
//In javascript - better  
shouldShowCustomiseBtn: computed('isBulk', 'didCustomise', {  
 get() {  
 let {  
 isBulk,  
 didCustomise,  
 isDeal  
 } = getProperties(this, [  
 'isBulk','didCustomise','isDeal'  
 ]);  
   
 return (isBulk && (didCustomise === false)) || isDeal;  
 }  
 }),  
  
//in Template - better  
{{if shouldShowCustomiseBtn}}  
 <button>Customise</button>  
{{/if}}

## Routing

### Route naming

Dynamic segments should be underscored. This will allow Ember to resolve promises without extra serialization work.

Example

// good  
  
this.route('foo', { path: ':foo\_id' });  
  
// bad  
  
this.route('foo', { path: ':fooId' });

### Always extend from BaseRoute

## Ember Data

### Be explicit with Ember Data attribute types

Even though Ember Data can be used without using explicit types in attr, always supply an attribute type to ensure the right data transform is used.

Example

// Good  
  
export default Model.extend({  
 firstName: attr('string'),  
 jerseyNumber: attr('number')  
});  
  
// Bad  
  
export default Model.extend({  
 firstName: attr(),  
 jerseyNumber: attr()  
});