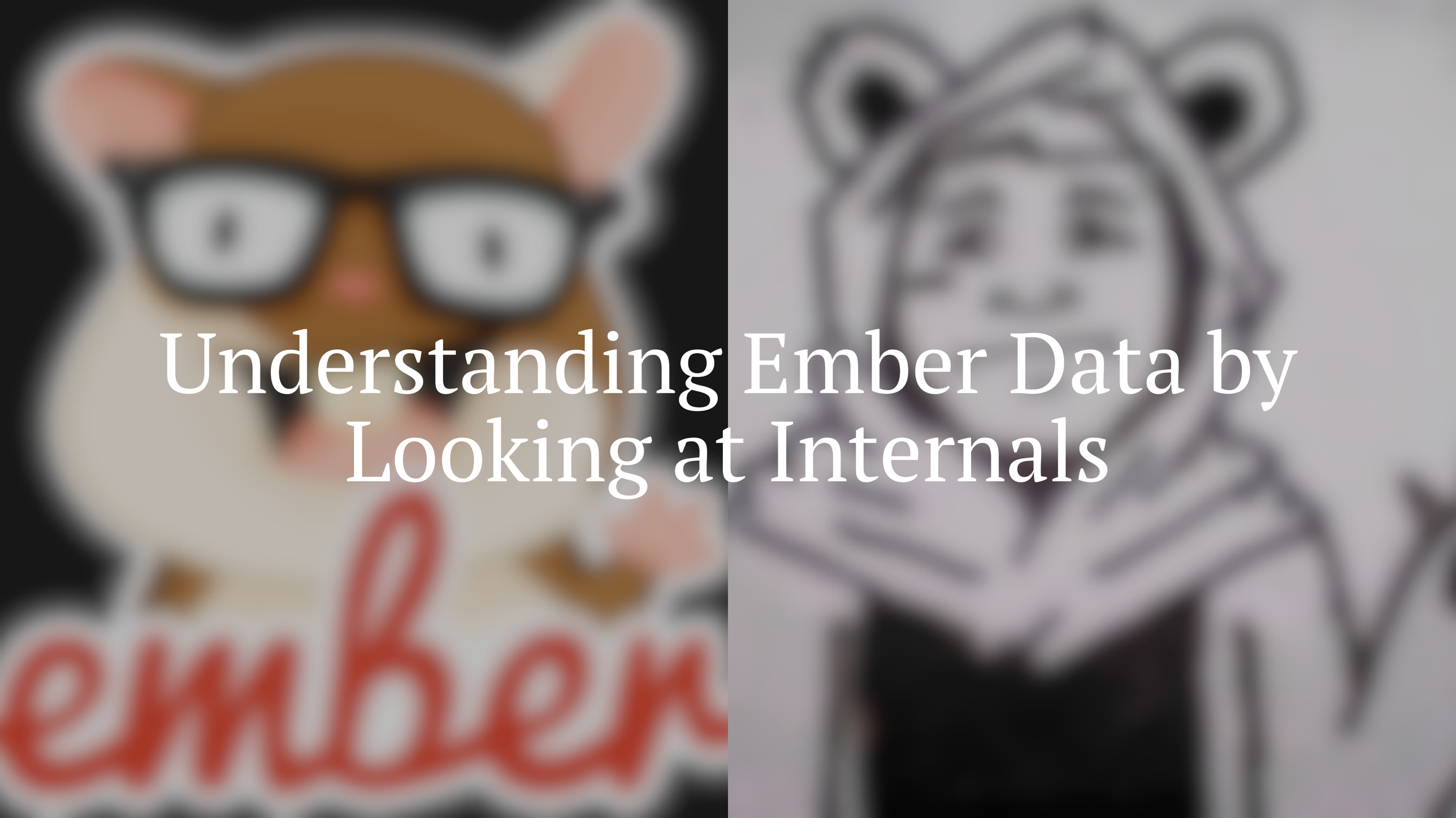


Understanding Ember Data by Looking at Internals

Ember Data is a library for robustly managing model data in your Ember.js applications.

Ember Data is magic

Ember Data is ~~magic~~ operating at a level of abstraction I'm not comfortable with yet

Disclaimer: Don't rely on private APIs if possible.

This stuff *will* change.

What It Isn't

— A relational database

— Rail's ActiveRecord

— The long sought after 1.0

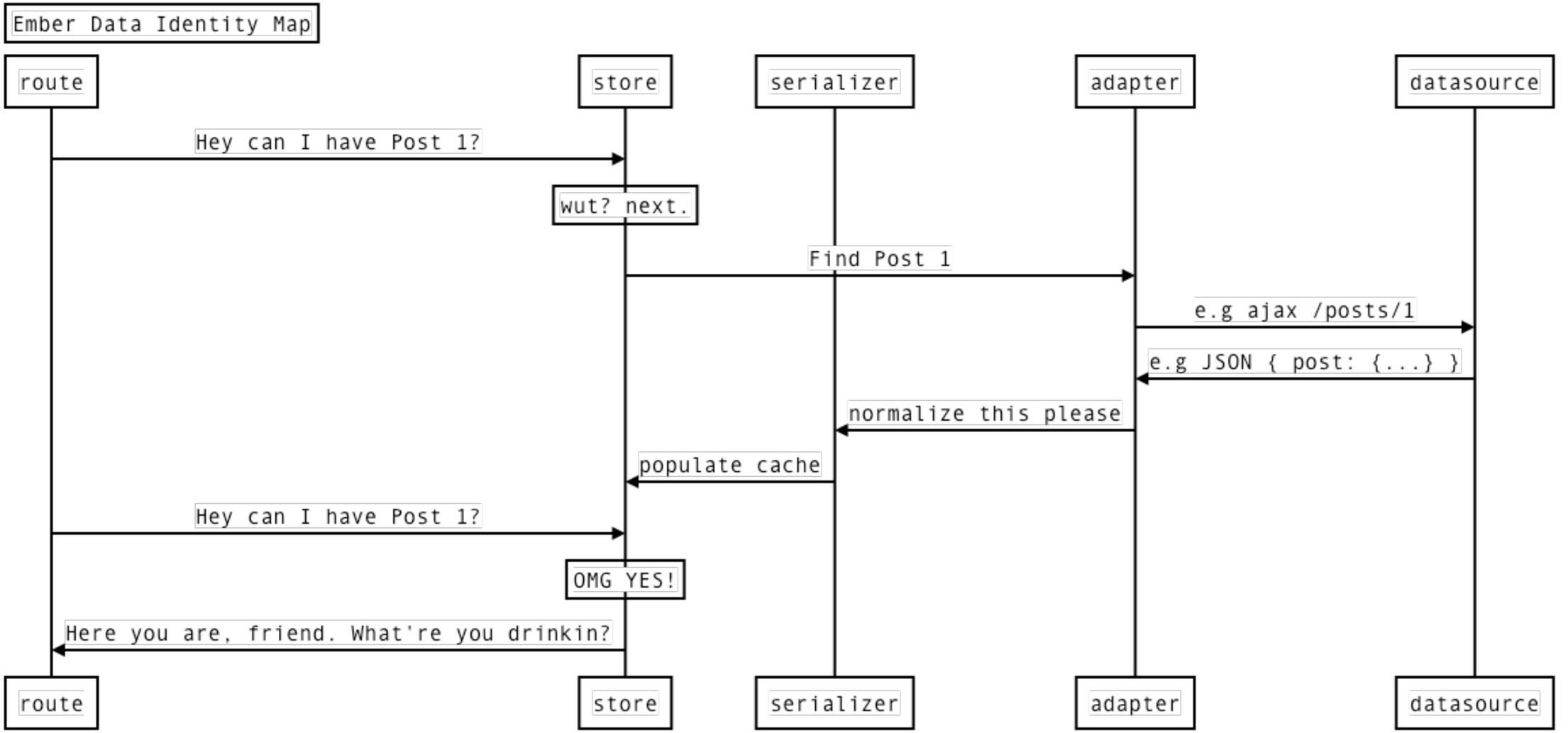
So What Is It?

— ORM "inspired"

— Beta, but used by many

— Identity Map / Cache

The identity map pattern is a database access design pattern used to improve performance by providing a context-specific, in-memory cache to prevent duplicate retrieval of the same object data from the database


Ember Data Identity Map
route
store
serializer
adapter
datasource
Hey can I have Post 1?
wut? next.
Find Post 1
e.g ajax /posts/1
e.g JSON { post: {...} }
normalize this please
populate cache
Hey can I have Post 1?
OMG YES!
Here you are, friend. What're you drinkin?
route
store
serializer
adapter
datasource

There are only two hard problems in Computer Science: cache invalidation and naming things.

— Phil Karlton

Mind the Cache

```
// skip the cache, then refresh it with response
store.find("post");
store.find("post", "cache-miss-id");
store.fetch("post", 1);

// refresh the cache
post.reload();
post.get("comments").reload();
store.push(data);
store.pushPayload(dataToBeNormalized);

// invalidate the cache
store.unloadRecord(post);
store.unloadAll("post");
store.unloadAll();
```

DS.Store

```
Store = Service.extend({

  /**
    @method init
    @private
  */
  init: function() {
    this._backburner = new Backburner([
      'normalizeRelationships', 'syncRelationships', 'finished'
    ]);
    this.typeMaps = {};
    this.recordArrayManager = RecordArrayManager.create({
      store: this
    });
    this._containerCache = Ember.create(null);
    this._pendingSave = [];
    this._pendingFetch = Map.create();
  }

  // ...
});
```

TypeMaps: The Store's Record Cache

```
// store.typeMaps =>
{
  // Guid enerated by type name
  "some-guid": {
    type: "post",
    // constant time access to all cached posts
    records: [{Post}, {Post}],
    // constant time access to a cached post
    idToRecord: { "postId": Post },
    // metadata from adapter
    metadata: { page: 1, numPages: 10 },
    // populated lazily by RecordArrayManager
    // upon first store.all func call
    // findAllCache: RecordArray
  },

  // ... type maps for other `DS.Model`s
}
```

```
// packages/ember-data/lib/system/store.js
//
buildRecord: function(type, id, data) {
  var typeMap = this.typeMapFor(type);
  var idToRecord = typeMap.idToRecord;

  var record = type._create({
    id: id,
    store: this,
    container: this.container
  });

  if (data) {
    record.setupData(data);
  }

  if (id) {
    idToRecord[id] = record;
  }

  typeMap.records.push(record);

  return record;
},
```

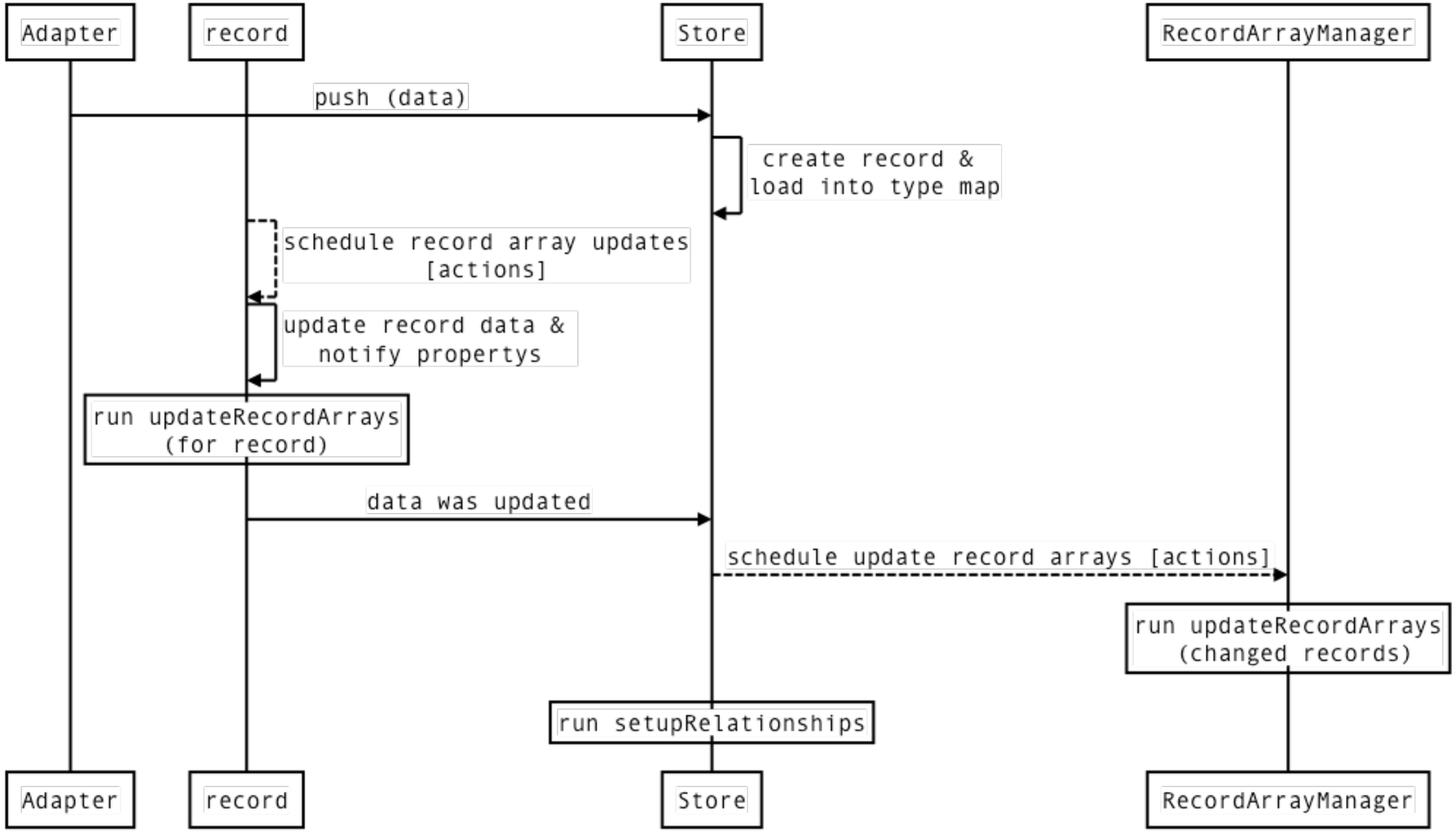
`RecordArrayManager`: Keeping Cached Slices in Sync

- `RecordArray(store.all)`
- `FilteredRecordArray(store.filter)`
- `AdapterPopulatedRecordArray (store.findByQuery)`

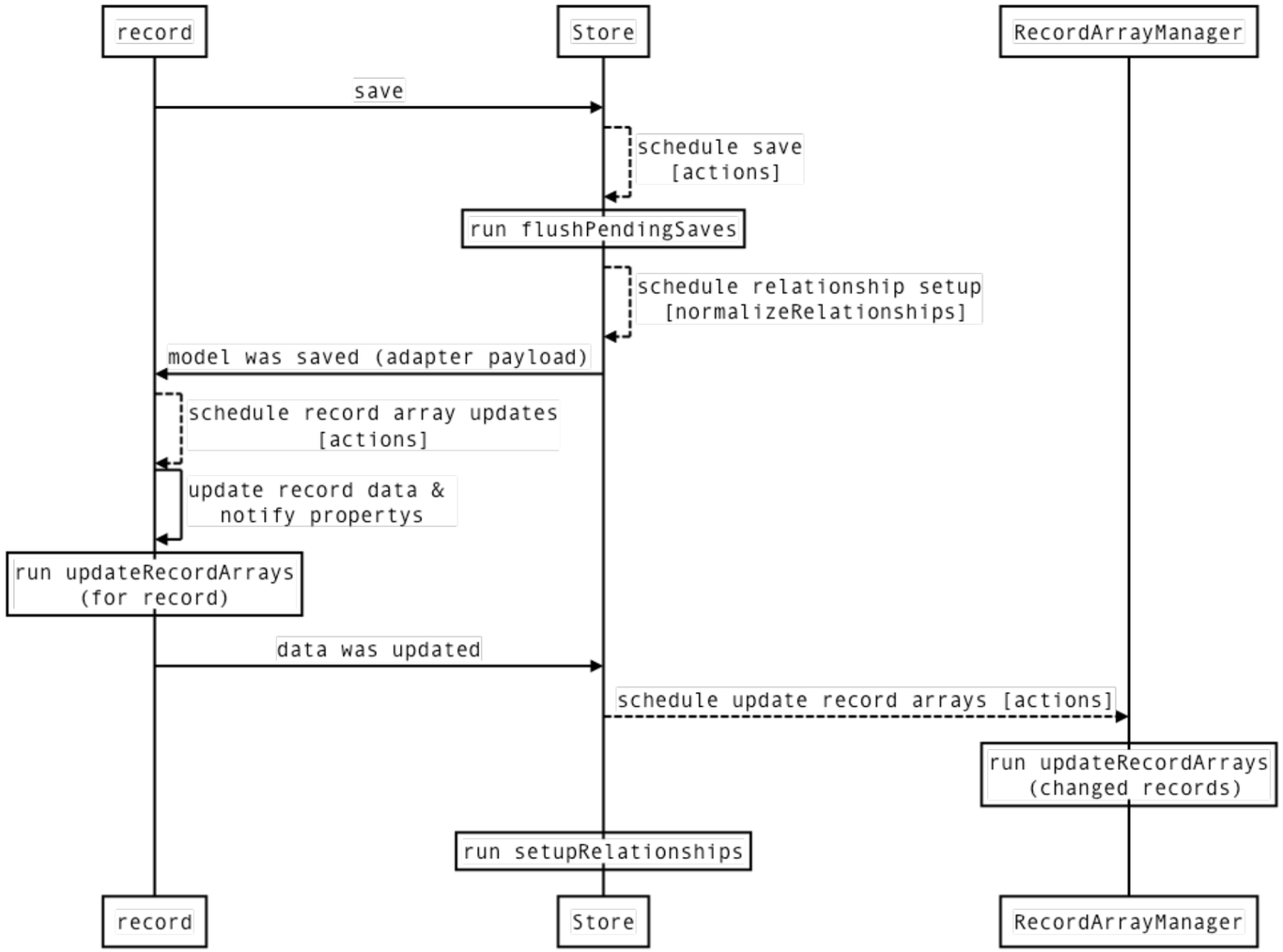
`RecordArrayManager`: Sync it!

- Has a map of record arrays by type called `filteredRecordArrays`
- Recompute's record arrays when data is loaded or unloaded into store
  - **Load**: Recompute record array's filter function
  - **Unload**: Rm record from manager's record arrays
- Also syncs model's internal `_recordArrays`
- **Caveat**: Can't currently GC record arrays (RFC)

Maintaining the Cache: Data Push



Maintaining the Cache: Read/Write to Datasource



Relationships

Ember Data Relationships

— Ember data's way of tracking dependencies between records
— You don't need (or can't have) all the data at once
— Relationships != Database Relationships

So What Is A Relationship Object?

```
//packages/ember-data/lib/system/relationships/state/relationship.js
//
var Relationship = function(store, record, inverseKey, relationshipMeta) {
  this.members = new OrderedSet(); // (1)
  this.canonicalMembers = new OrderedSet(); // (2)
  this.store = store;
  this.key = relationshipMeta.key;
  this.inverseKey = inverseKey; // (3)
  this.record = record;
  this.isAsync = relationshipMeta.options.async;
  this.relationshipMeta = relationshipMeta;
  this.inverseKeyForImplicit =
    this.store.modelFor(this.record.constructor).modelName + this.key; // (4)
  this.linkPromise = null;
  this.hasData = false;
};
```

Making our Models Aware

```
import DS from "ember-data";

// app/models/post.js
//
export default DS.Model.extend({
  comments: DS.hasMany("comment"),
});

// app/models/comment.js
//
export default DS.Model.extend({
  post: DS.belongsTo("post"),
});
```

Expanding Relationship Macros

— At the time of extending, expands `DS.hasMany` and `DS.belongsTo` into computed property getter/setter.

```
//  app/models/post.js
//
export default DS.Model.extend({
  comments: DS.hasMany("comment"),
});

// is (roughly) turned into...
```

```
// app/models/post.js
export default DS.Model.extend({

  comments: Ember.computed({

    get: function(key) {
      var relationship = this._relationships[key];
      return relationship.getRecords();
    },

    set: function(key, records) {
      var relationship = this._relationships[key];
      relationship.clear();
      relationship.addRecords(records);
      return relationship.getRecords();
    }

  }).meta({type: 'comment', isRelationship: true, options: {},
           kind: 'hasMany', key: "comments"})
});
```

Modeling your Data

— How much of it do you need?
— How quickly does it get stale?
— What dependencies does it have?
— Do you control the client? server? both?

Yet Another Cliche Example

```
// app/models/post.js
export default DS.Model.extend({
  comments: DS.hasMany("comment", { async: true }),
  author: DS.belongsTo("author"),
});

// app/models/author.js
export default DS.Model.extend({
  posts: DS.hasMany("post", { async: true })
});

// app/models/comment.js
export default DS.Model.extend();
```

Async: true

```
// app/models/post.js
export default DS.Model.extend({
  firstComments: DS.hasMany("comment"),
  restComments: DS.hasMany("comment", { async: true })
});

// for has many...
post.get("firstComments") // => ManyArray (of comments)
post.get("restComments")  // => PromiseManyArray (resolves with comments)
    .then(function(comments) { ... });

// and similarly, belongs to...
comment.get("post")       // => post
comment.get("asyncPost") // PromiseObject (resolves w/ post)
       .then(function(post) { ... });
```

Implicit Model Relationships

Note: Implicit relationships are relationship which have not been declared but the inverse side exists on another record somewhere

```
export default DS.Model.extend({ // app/models/post.js
  comments: DS.hasMany('comment')
})
export default DS.Model.extend({}); // app/models/comment.js

// e.g
comment.destroyRecord().then(function() {
  Ember.assert("Post doesn't hang onto deleted comment",
    !post.get("comments").contains(comment);
  );
});
```

Record Initialization

```
// packages/ember-data/lib/system/model/model.js
// called on init
//
_setup: function() {
  // ... DS.attr stuff ...
  this._relationships = {}; // (1)
  this._implicitRelationships = Ember.create(null); // (2)
  var model = this;
  this.constructor.eachRelationship(function(key, descriptor) {
    model._relationships[key] =
      createRelationshipFor(model, descriptor, model.store); // (3)
  });
}
```

```
post._relationships["comments"]
// ManyRelationship
//   members / canonicalMembers
//   record: post
//   key: "comments"
//   inverseKey: "post"
//   manyArray: [comment]

comment._relationships["post"]
// BelongsToRelationship
//   members / canonicalMembers
//   record: comment
//   key: "post"
//   inverseKey: "comments"
//   inverseRecord: post
```

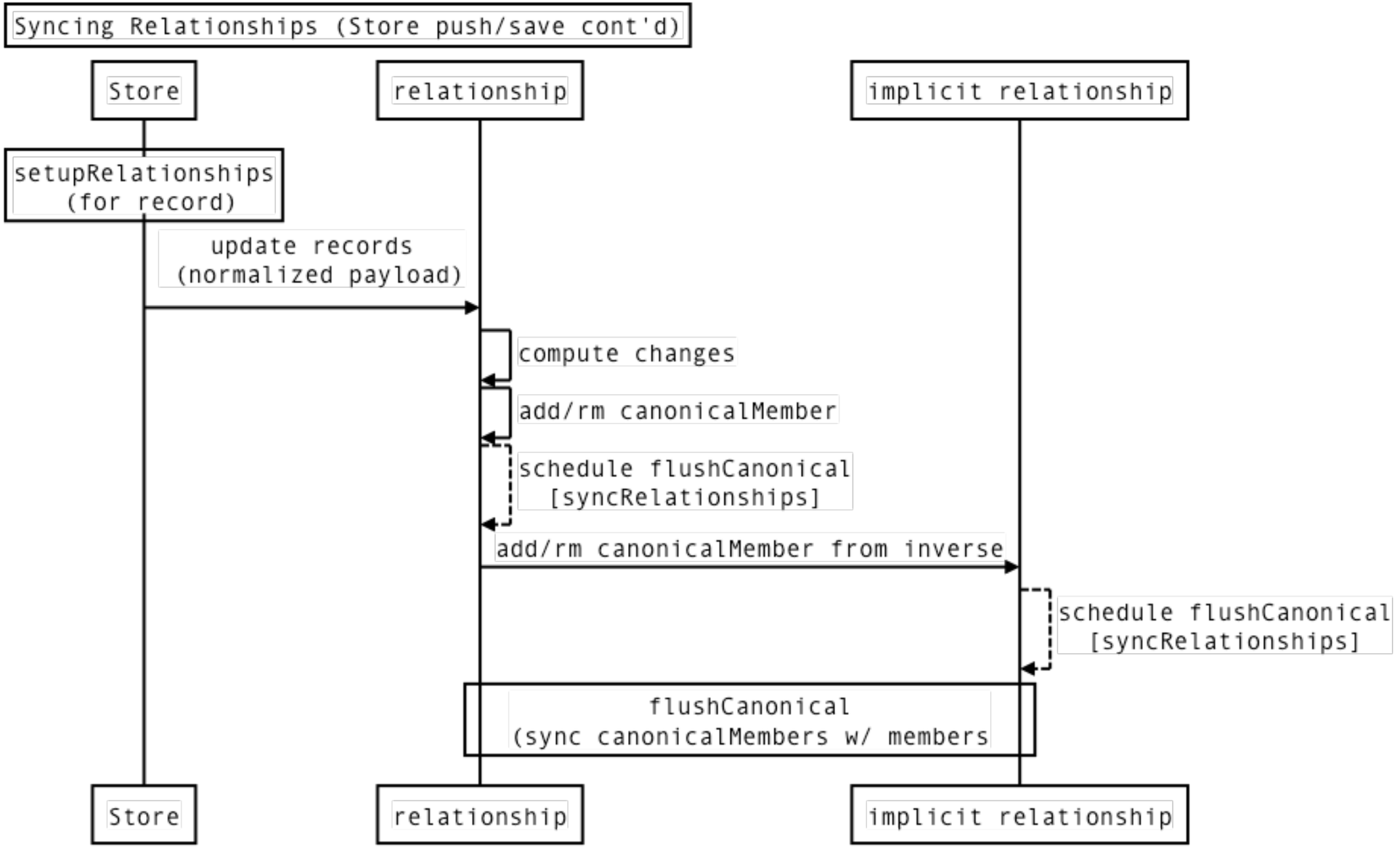
Keeping Both Sides in Sync (pseudo-code)

```
// (1) get the post's comments relationship
commentsRel = post._relationships["comments"];

// (2) add comment to post's comments
commentsRel << comment // (2)

// (3) find "other side" of relationship via inverse
// Note: it was specified or inferred by the hasMany macro
// in our Post model. In this example, it is "post"
postRelKey = comment._relationships[commentsRel.inverseKey]

// (4) Set post as comment's post (updating belongsTo)
comment._relationships[postRelKey] << post // (3)
```


Syncing Relationships (Store push/save cont'd)
Store
relationship
implicit relationship
setupRelationships
(for record)
update records
(normalized payload)
compute changes
add/rm canonicalMember
schedule flushCanonical
[syncRelationships]
add/rm canonicalMember from inverse
schedule flushCanonical
[syncRelationships]
flushCanonical
(sync canonicalMembers w/ members
Store
relationship
implicit relationship

The beauty of magic is that it was right in front of you the whole time.

Disclaimer: My understnanding of how ember-data works, not how you should use it.

Please use the guides on emberjs.com

Thanks!

- Questions or Feedback?
- Slides will be on Speakerdeck
- Markdown (with notes) will be on Github
- Links:
  - https://github.com/tonywok
  - https://speakerdeck.com/tonywok

Tony Schneider / @tonywok