Pique Design

Sammie Jiang, Maria Strasky, Reid Oliveira, Erik Dolinsky

Introduction

We plan to develop Pique as an application that social media consumers can use to access top and trending content from multiple sources with minimal latency in one simply executed session. This requires a well-designed, visually pleasing front-end, and a highly efficient back-end.

This document outlines the architecture, high-level design, and detailed design of the Pique system.

System Architecture and Rationale

The Pique system architecture is based on the Model View Controller Application model, where the data within the model is populated asynchronously by a Pipe and Filter sub-architecture.

Filter: Transforms the data it receives and hands the filtered data off to the next component

Pipe: Transfers data between components

Model: Manages data and system interaction logic

View: Displays output representation of data

Controller: Processes input and translates into commands for either the model or view

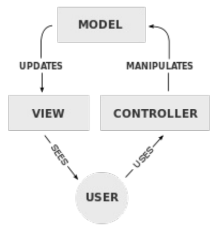


Image source (https://en.wikipedia.org/wiki/User:RegisFrey)

MVC Architecture (https://developer.chrome.com/apps/app_frameworks)

Pipes and Filters Pattern (https://msdn.microsoft.com/en-us/library/dn568100.aspx)

Overview

The major components of our system are as follows:
1. Website to display top and trending social media posts
2. Backend that organizes the posts and serves curated content
3. Data cache to hold streams of recent post data and queues of sorted data
4. Data collectors that gather posts from major social media sites via their APIs (Twitter, Reddit, Imgur, etc.)

Components

Play

Play Framework (https://www.playframework.com/)

The web application is built using Play, an open source web application framework for Java and Scala. It is built on top of Akka, a framework built around using high level abstractions and having high fault tolerance. Play extends this to the web application domain to focus on making lightweight and stateless web applications that are easily changeable and adaptable. Play features RESTful API hosting, website templating in Scala (via the Twirl Template Engine), MVC application model, and *"services"* Play's version of always-on worker processes.

Redis

Redis (http://redis.io/)

Redis is an open-source on-memory data structure store, which can be used as a database, cache, or message store. It supports automatic eviction (Least Recently Used), persistence to disk, automatic failover, keys with limited time-to-live, multiple queue streams, and the ability to publish or subscribe to specific streams. Open-source libraries exist to interface with Redis in multiple languages, including Java and Scala.

External APIs

Imgur API (https://api.imgur.com/) Twitter API (https://dev.twitter.com/rest/public) Reddit API (https://www.reddit.com/dev/api/)

The data for our system is provided by other social media applications such as Imgur, Reddi, and Twitter. APIs for these platforms tend are RESTful interfaces that allow us to query for publicly available posts and their related information. Other additional social media sources may have non-REST APIs and should be supported by our system's generic types if needed.

Bootstrap

Bootstrap (http://getbootstrap.com/)

Bootstrap is a front-end framework for HTTP, CSS, and JavaScript designed to streamline front-end development. It boasts dynamically adjusted layout, making web applications responsive to the client device's characteristics.

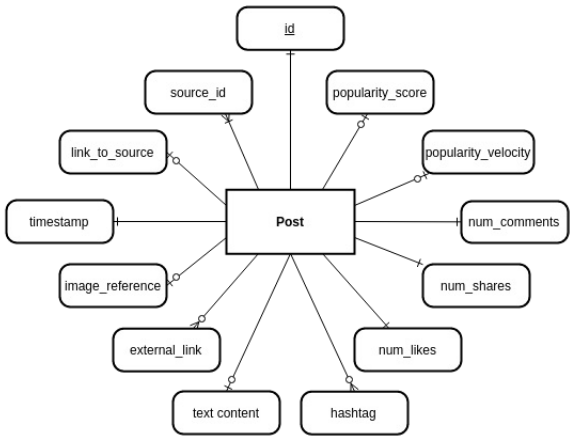
Amazon Web Services (AWS)

AWS (https://aws.amazon.com/)

Amazon Web Services provides a variety of cloud-based services to fulfill a variety of computing needs. We'll be using AWS' Elastic Beanstalk application hosting service. This will allow us to host Pique on a remote server for use by all users. Elastic Beanstalk also provides one-step deployment, which we can combine with Play's one-step build process for easy deployment to production.

Data

Pique operates on a single non-relational set of individual data entities, and thus maintains a simple Entity-Relationship model, as follows:



The non-relational nature of Redis allows for any Post entity to be stored in one or more channels of data, as we see fit. More information about the structure of these data channels is found in “Detailed Design.” A unique identifier will be associated with each entity.

Since a Post entity could come from any one of Twitter, Reddit, or Imgur, we shall use “likes,” “shares,” and “comments” as generic terms for user interactions with a post, even though — for example — on Twitter they may be called “favorites,” “retweets,” and “comments” instead.

Posts that have been gathered but not yet stored will not yet contain a popularity score or popularity velocity. The number of comments, shares, likes, and timestamp (of content creation) will be used to create a popularity score, which will allow us to sort top posts. Evaluating a post’s popularity score over time will yield a popularity velocity, which can be used to sort trending posts.

Sorted Posts containing popular hashtags will be cached separately by hashtag, allowing users to search for posts by hashtag.

The remaining information contained in a Post entity will be used by the front-end to display the content and related information of the post among other posts.

In order to serve data to the front end, Posts to be displayed on the front end will be stored in a PostList entity, which is simply a list of PostLists. This allows for a very simple paging mechanism when displaying sections of posts.

Protocol Buffers (Protobufs)

Protobufs (https://developers.google.com/protocol-buffers/)

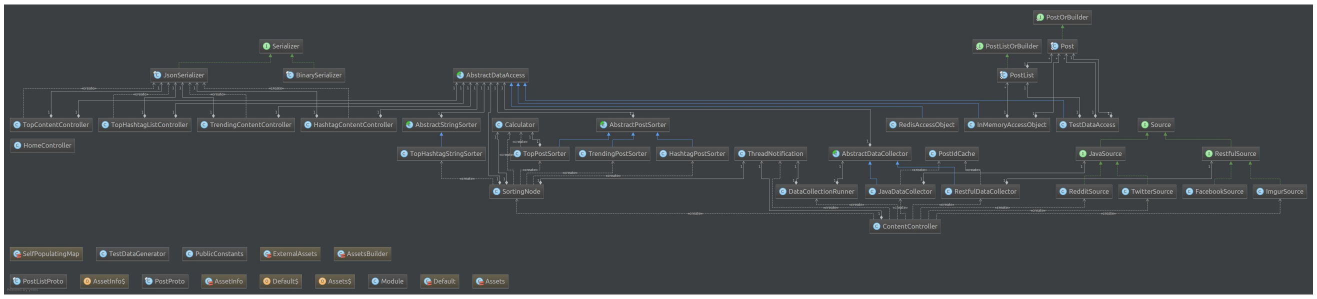
Protocol Buffers is a method of serializing structured data, developed by Google, designed to be smaller and faster than XML. Extensible data structures are converted to binary data streams, allowing for minimal required storage and fast transmission between components. Post entities will be stored in protobuf format throughout the Pique system.

Detailed Design

This section outlines the details of each component of the Pique architecture in more detail. For an even more detailed design of specific components, see the corresponding **Detailed Design** section.

Class Diagram

The class diagram for our system is shown here:



** *This will be extended with the implementation of additional features*

Components

A description of how each component will fit into the system, and some information about its responsibilities within thePique architecture.

Data Consumer

There will be a class that will communicate with each of the external social media APIs to fetch new data. We are planning on implementing basic functionality to consume content from Twitter, Reddit, and Imgur, but a general class structure will allow us to extend to other services in the future.

Redis

A server running Redis will be used for two different data store models within the Pique architecture, as follows:

- **Post Stream:** A server running Redis will store newly found content in a set of queues, each corresponding to a source site.
- **Content Display Queue:** A set of posts sorted by popularity index or trending index, to be generated by the sorting node.

Sorting Node

A Play framework service which pulls posts from multiple post streams and evaluates content based on similarity and available popularity metrics, or by a specific hashtag keyword.

Backend API

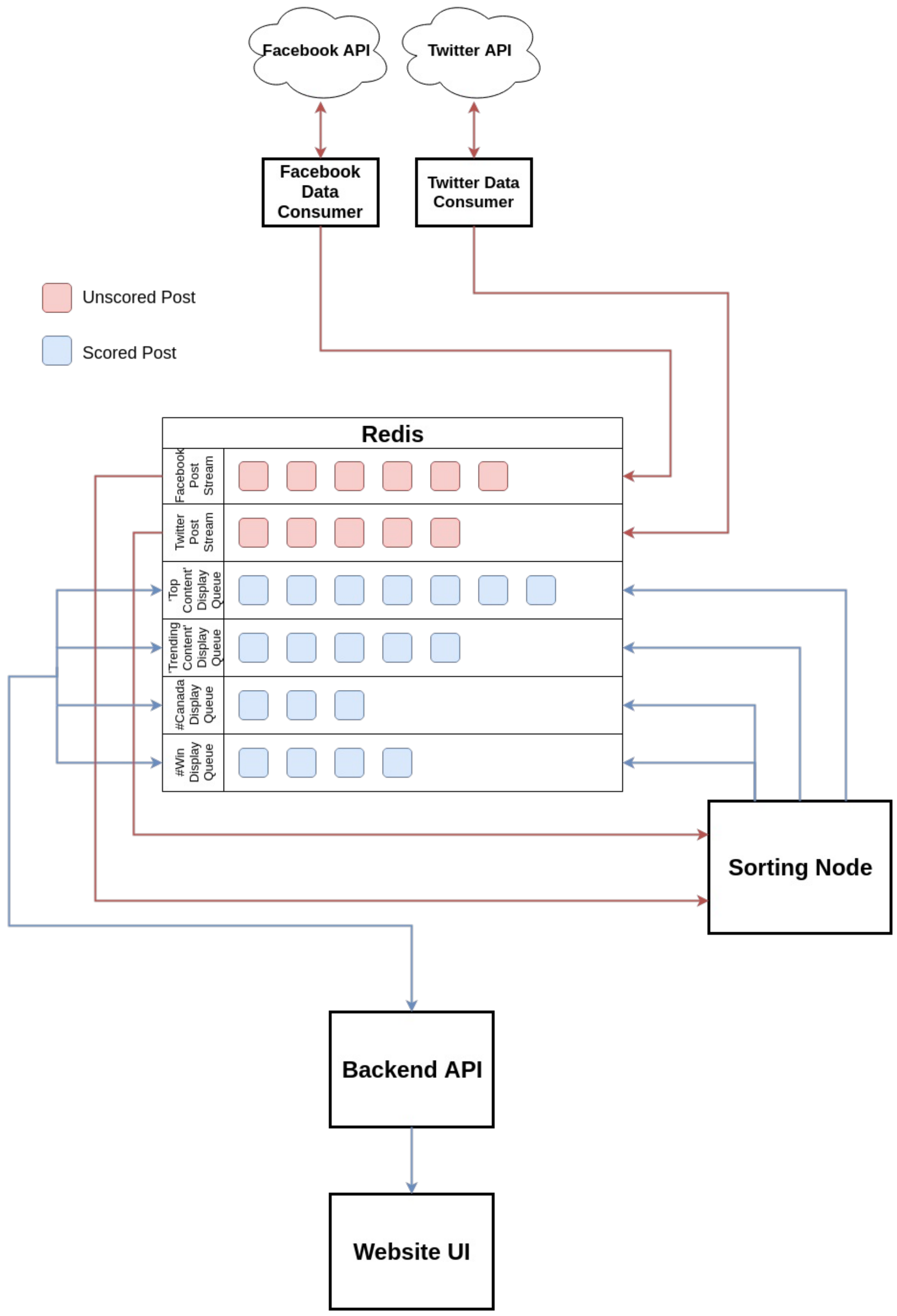
A class containing basic functionality to mediate between the front-end and Redis server and fetch post data when requested.

Website UI

This class will contain the main UI that will display the posts, and let the user sort them according to three specific views: top, trending, or hashtag.

Detailed Data Flow Diagram

The following diagram shows the flow of data — both requests and data entities — throughout the Pique architecture.

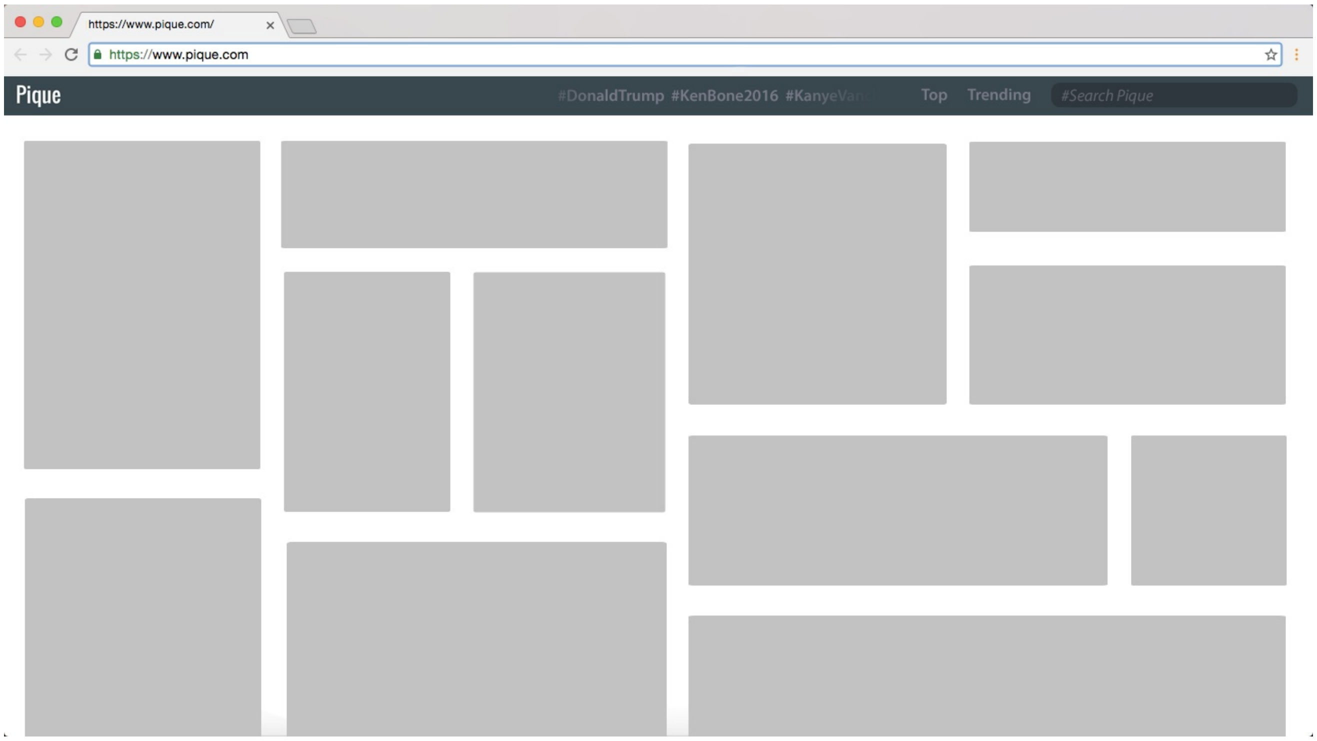


GUI

The user interface is built using Scala-based HTML Templates, using the Twirl Template Engine (which is built into the Play Framework), and uses the Bootstrap Framework.

Basic Feed View

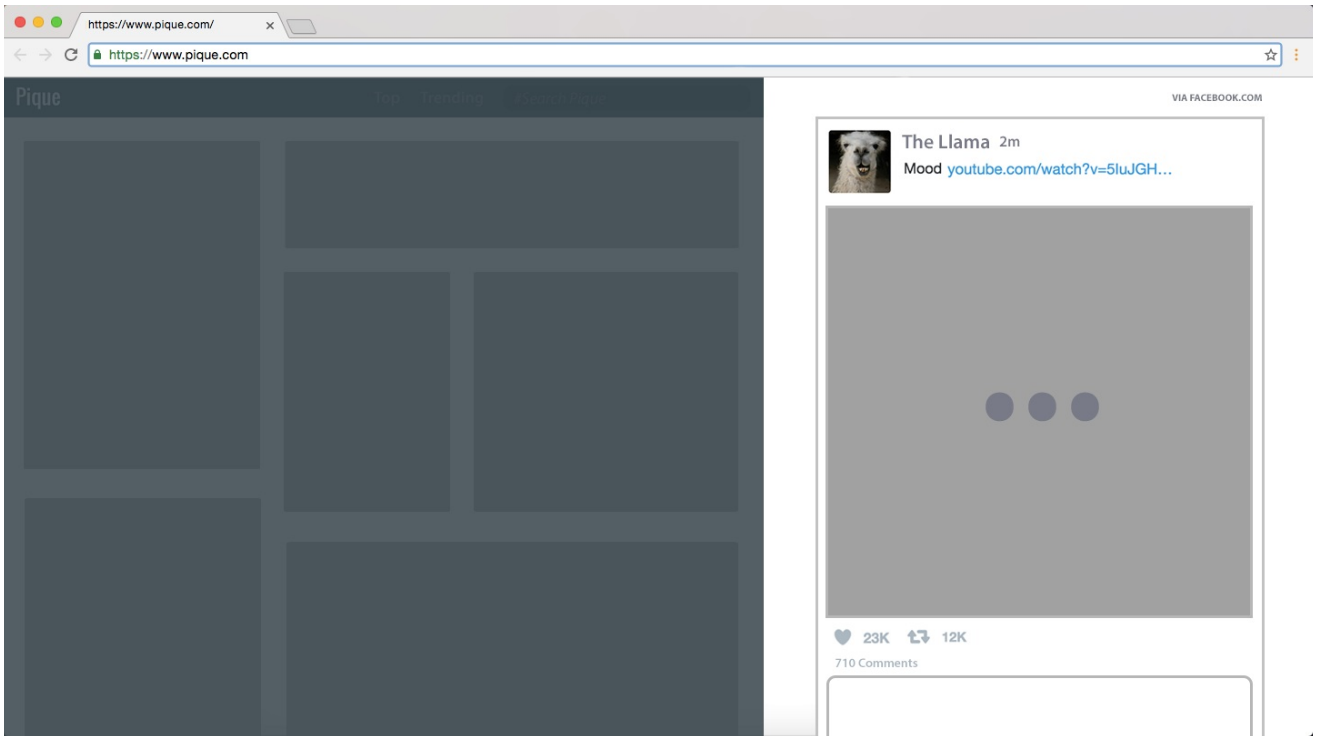
This is the view the user will see upon loading the landing page.



Top content will be displayed by default, but the user can toggle between 'Top' and 'Trending' content by clicking the respective fields at the top right hand menu bar, or search for specific hashtags in the search bar. A scrolling list of top hashtags is displayed to the left of these options, and can be clicked to load content containing the particular hashtag.

Expanded Posts View

When a user clicks on a post to access it in more detail, the post expands on the right hand side of the window, and the remaining content is compressed to the left hand side of the window, as below:



Validation

Validation is baked into every stage of the Pique engineering process. All documents created, design plans, and implementations are peer-reviewed before a preliminary release and at every iteration. During this review process, the content developed is compared to the requirements of the user, as well as the requirements we have set out for the system in our *Requirements Vision and Scope* Document.

When unsure about a particular design decision, we will verify our choices with potential users of the product, or technical peers depending on the nature of the problem.

As Pique development is iterative, prototypes will be produced throughout development. We can leverage these prototypes to test our design decisions with the user.

Pique development follows a fairly strict feature branch workflow, with peer reviews at multiple steps of design and implementation. For more information about this process, see the *Test Plan* document.

Data Tier

This section outlines how we've chosen to store and transmit data throughout the Pique architecture.

From the start, we knew we needed to be able to store a large amount of data, and that we needed to access it very quickly. Since Pique's goal is based around getting trending content to the user quickly, we found that we could take some liberties with data persistence. It's also not the end of the world if we lose some data; if it's important, pique will see it again.

Data Storage

Within Pique, data is stored and transferred in a relatively common format, whether as Java Objects, or serialized into byte arrays. The method of data storage is also abstracted, allowing us to choose the best method of storage based on whether we are running Pique in production, test, or development environments. This section outlines our choices of data storage and transmission throughout Pique.

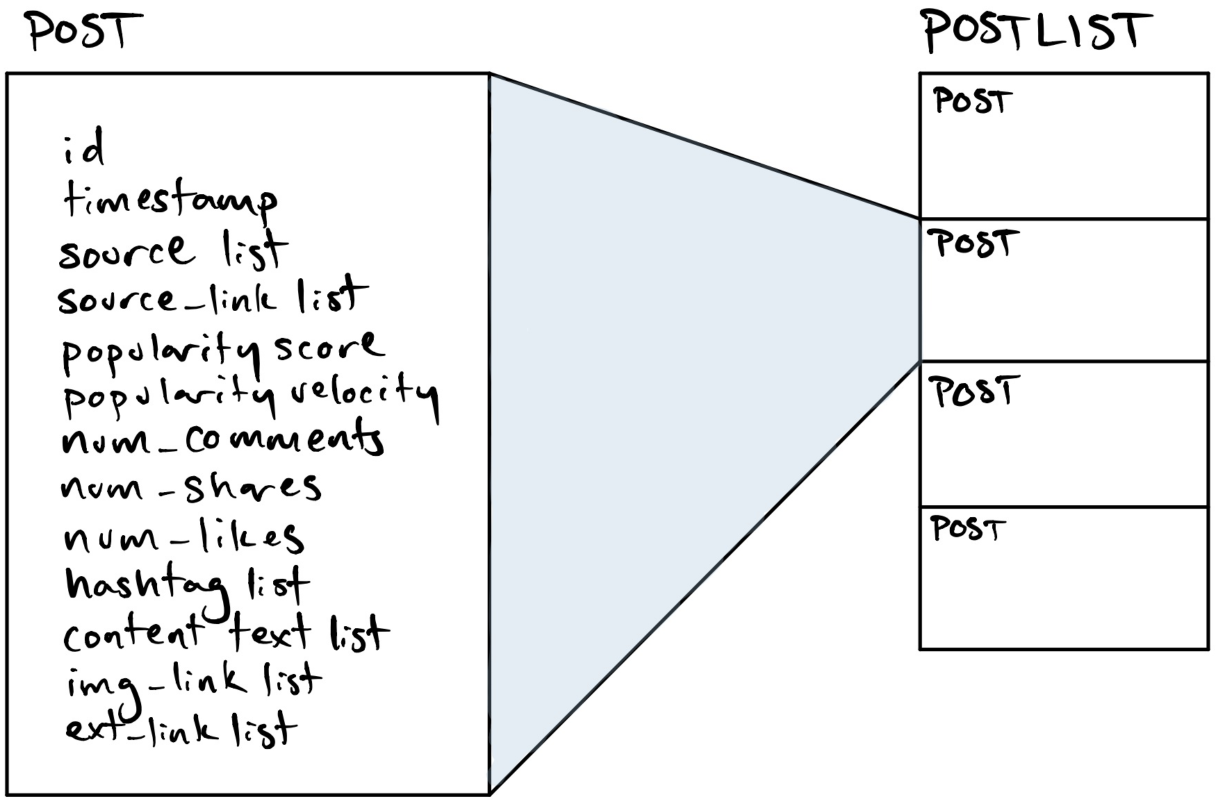
Protocol Buffers

Protocol buffers (https://developers.google.com/protocol-buffers/), or protobufs, are an extensible mechanism for serializing structured data, designed by developers at Google. Just like XML or JSON, data can be stored in an extensible structured format. However, unlike XML or JSON, protobufs are serialized into an array of bits, making them incredibly small. This makes protocol buffers perfect for Pique, as they are light on storage, and can be transmitted very quickly.

How we use Protobufs

Pique uses two different, but very closely related classes of protocol buffers. Each post gathered contains a particular set of data, which must be stored. We've created the aptly named **post** entity to store such information for each post, contained in its own buffer.

Pique needs a way of sending these posts to the user's browser. Unfortunately, protobuf parsing tools can't distinguish between individual elements in a bit stream, so the simplest way is to construct a second protobuf type, which contains a list of **post** entities, which we've called **postlists**. Below is a diagram showing the relationship between the two, and the type of data we've chosen to store in a **post**.



Pique uses these protocol buffers not only as a method of storage, but can also use them as serialized data served to the front-end or some other subsystem which needs to access the same data a user would.

Abstraction in Data Storage and Transmission

The implementation abstracts interaction with data stores, including the type of data that is stored. In the event that we decide to swap out data stores, or the method in which we store data, we would simply need to write a new class with the required methods to do so.

The same can be said about serialization. Pique currently serializes data to the front end in JSON format, but if we want to return data in protobuf binary or a different format, we simply need to edit our data-serving controllers to accept a new `Content-Type` header type, and add a simple serializer class to translate the stored data to the required data type.

Serializer

The `Serializer` Interface provides a template for what a serializer should do. In short, it should be able to process PostList objects, and translate them into a format for transmission to a browser or REST client. Currently, we have two separate implementations of `Serializer`.

BinarySerializer

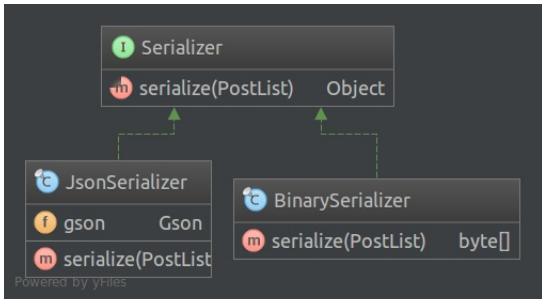
Originally, we had hoped to implement the front-end in such a way that it could deserialize posts from Protobuf binary wire format. However, after further research into other developers' past experiences with using Protobuf for serving front-end (and much trial and error), we have decided not to serve the front-end using Protobuf wire format at this time. We've kept it in our implementation, as it may prove useful with further iterations of the product.

JsonSerializer

`JsonSerializer` transforms PostLists into JavaScript Object Notation, using Google's gson library (https://github.com/google/gson). JSON is a natural choice for serving a JavaScript front-end.

Class Diagram

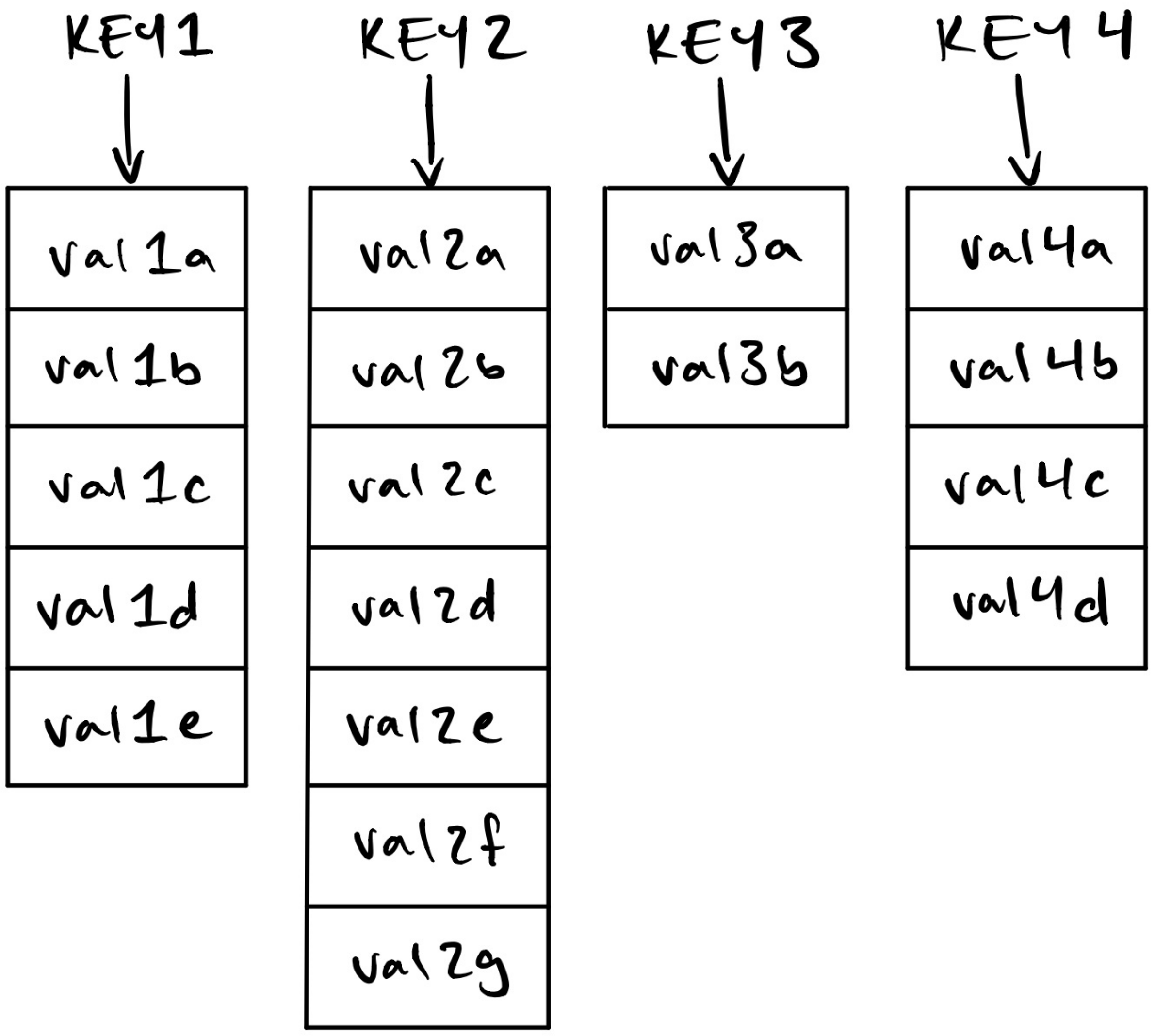
The following diagram shows the fields, methods, and relationships of all Serializer classes:



Redis

Data is really cool. Fast data is even cooler. Redis (http://redis.io/) is a **data cache** *(not database)* that allows for blazingly fast data access, and what we've chosen to use to cache our data in Pique.

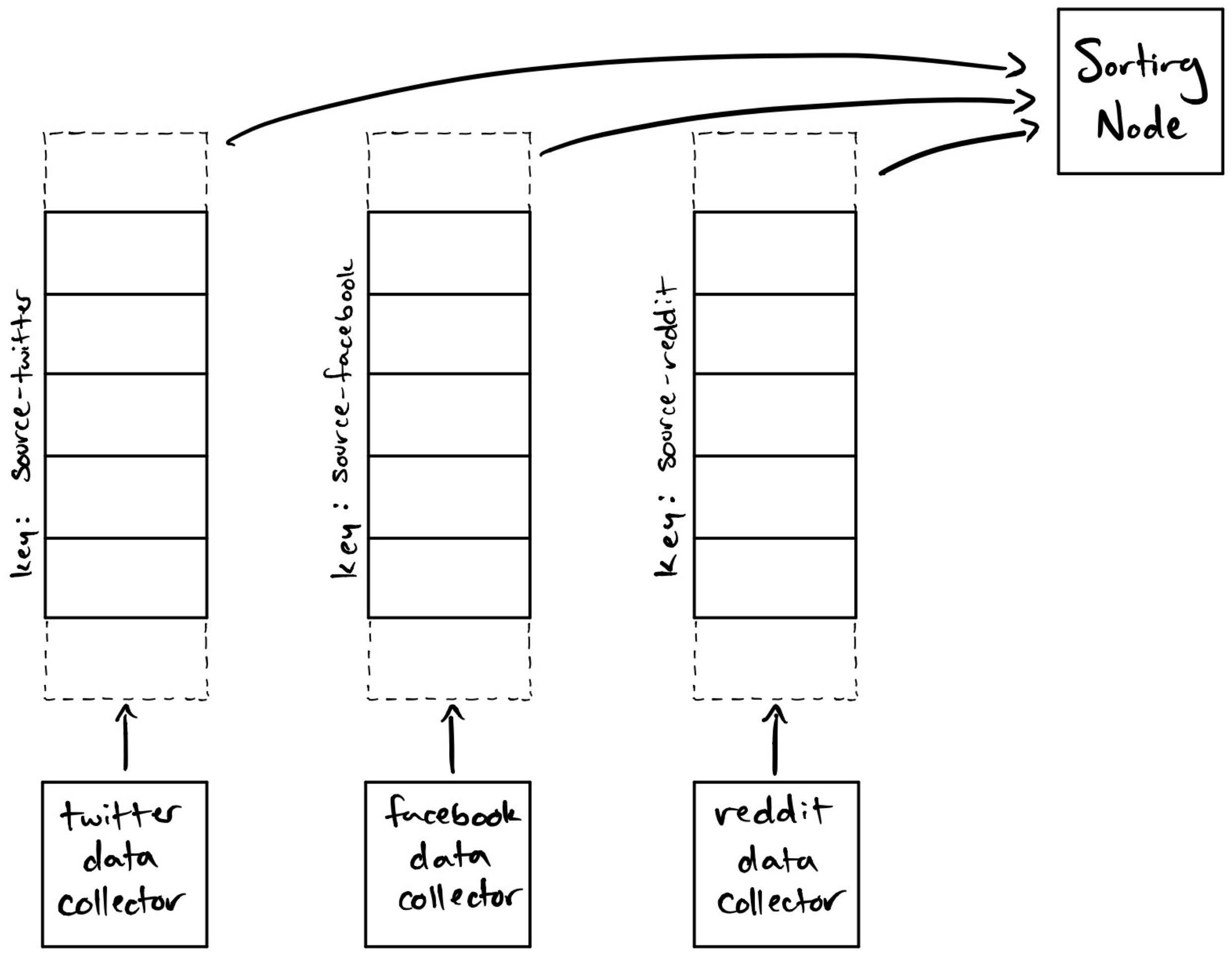
Redis is fast mostly because it is implemented to use a computer's memory, not its hard drive. It operates on a key-value store, much like a dictionary, with a few extra features, like data expiry, automatic failover, and optional persistence to disk. Redis' keys map to lists of values by default, as shown below.



How we use the Data Tier

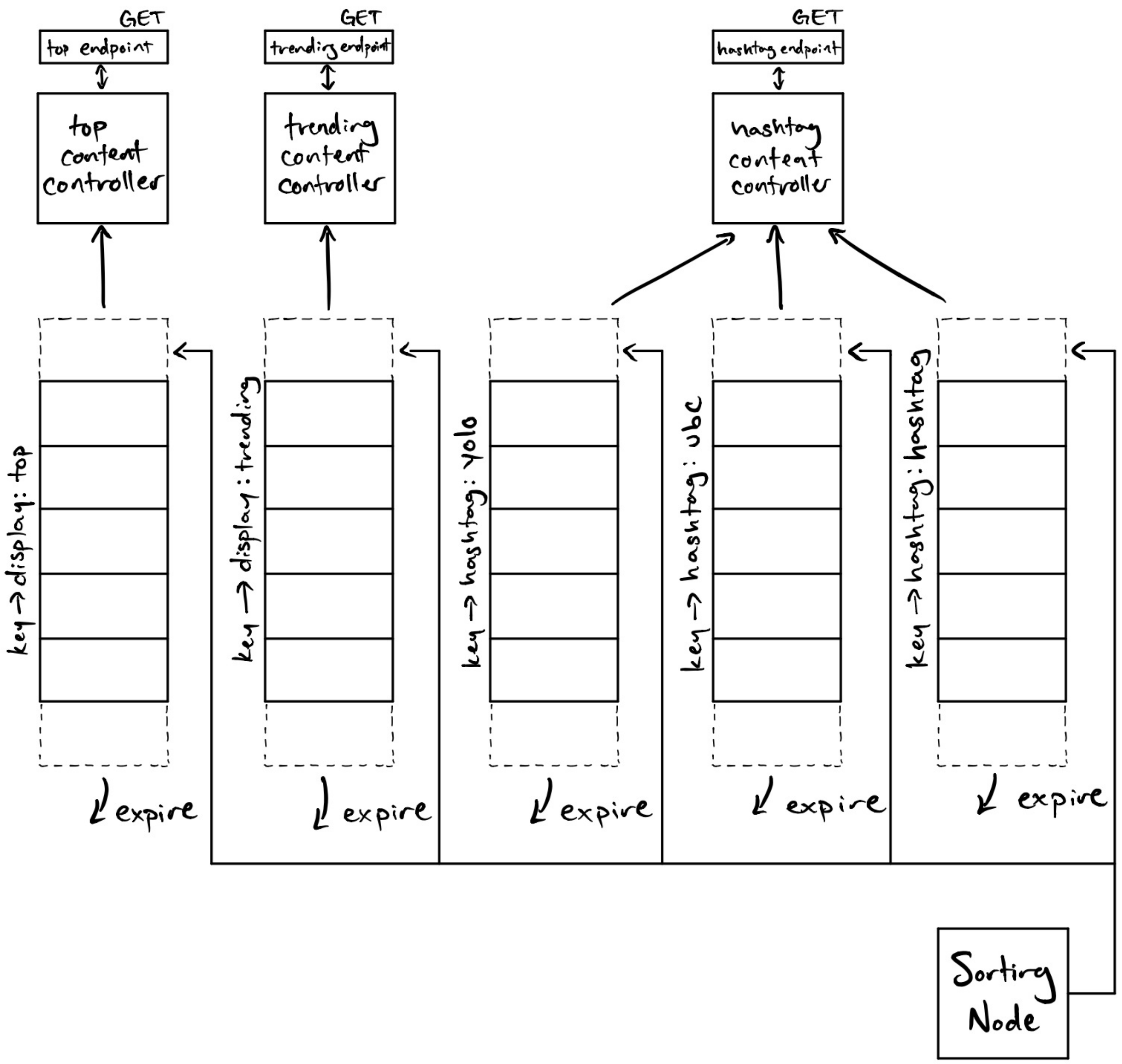
The Data Tier is used in two ways within Pique. One is essentially a set of first-in first-out (FIFO) queues, and the other is a set of stacks where nothing really gets popped off the top of the stack, and PostLists disappear periodically from the bottom of the stack (but we want that to happen).

First up, the queues. When data is gathered from social media sites, a **post** is created loaded into a particular queue associated with the source. The source name (prefixed by our selected source namespace) serves as the key within Redis, and the value list formulates the queue. The sorting node then consumes the posts off the other end of the queue, as shown below.



It’s important to note that the set of source content queues is arbitrarily large, and that *facebook*, *twitter*, and *reddit* are used here for example. For the final deliverable, we will have Twitter, Reddit, and Imgur implemented. See our **Requirements** document for more details.

The sorting node creates **postList** entities, which end up being the list of content the viewer sees at the front-end. Each **postList** is loaded into the *front* of the list corresponding to a particular Redis key (again, with a particular namespace prefix), so that the controllers always access the most recent content. The oldest **postList** in a particular key-value list expires when the list reaches a certain size.



Data Expiry

In the case of our **PostList** stacks, we've decided to implement expiry on both stored **postList** entities within any key, as well as key expiry. This functionality is currently implemented in `RedisAccessObject` only, for use in a production environment.

PostList Expiry

Within any key, if the stack of **PostList** entities reaches a certain size, the stack will be truncated from the bottom when a new **PostList** is added to the top of the stack. This helps manage our storage space, and ensures that outdated data does not affect our calculations in sorting & ranking.

Key Expiry

When a key is created, updated, or accessed, a time-to-live value for the key is created. If the key is not updated or accessed again before the time limit is up, it will expire and be removed from storage. This is particularly advantageous when dealing with hashtags, as unpopular hashtags will be removed from our system, so that the user only obtains relevant and recent data, and Pique will see better storage performance over time.

Namespaces

Pique uses namespaces to distinguish different families of keys within its data store(s). Namespaces are delimited from key names using a single ':'. The Namespaces used in Pique are as follows:

Namespace	**Definition**
source	Keys under this namespace contain Posts generated by a data collector
display	Keys under this namespace contain PostLists generated by the sorting node, for display on the front-end
displaystring	Keys under this namespace contain Strings sorted by the sorting node, for display on the front-end
hashtag	Keys under this namespace contain PostLists (for display and further processing) containing Posts with a specific hashtag, as denoted by the key name
test	Keys under this namespace contain data generated by a test. Tests should clear out these keys when finished

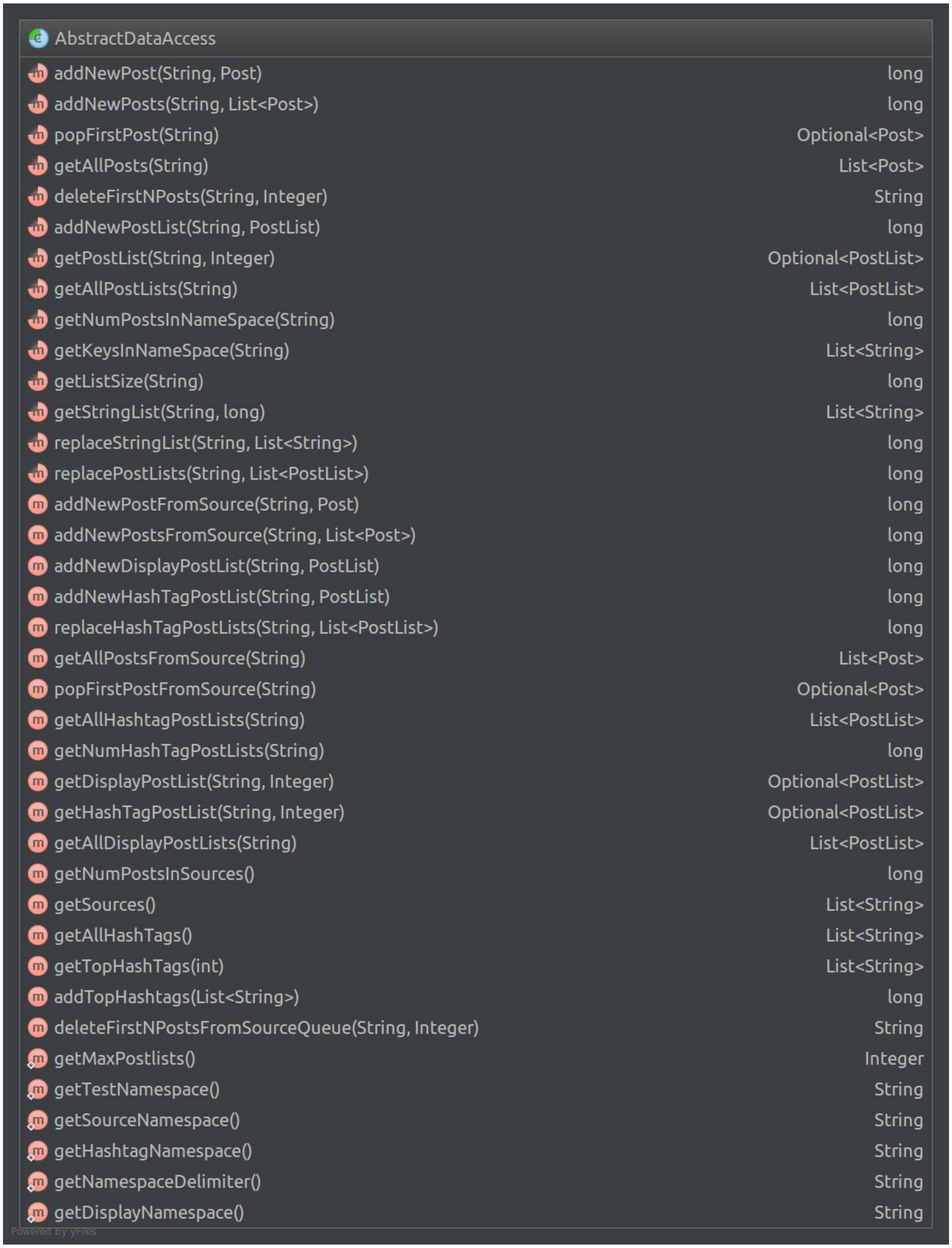
Data Access Classes

Pique employs abstraction in its data access. This gives us the freedom to swap out our storage format, or even entire data storage service, with minimal work.

The main Data Access class, `AbstractDataAccess` defines the functionality for our data tier. Classes that extend this class are required to communicate between the Pique backend and our method of data storage. They translate `String` keySpaces and `Post` and `PostList` value lists to and from the data storage format.

Functionality

`AbstractDataAccess` supports the following methods:



AbstractDataAccess	
addNewPost(String, Post)	long
addNewPosts(String, List<Post>)	long
popFirstPost(String)	Optional<Post>
getAllPosts(String)	List<Post>
deleteFirstNPosts(String, Integer)	String
addNewPostList(String, PostList)	long
getPostList(String, Integer)	Optional<PostList>
getAllPostLists(String)	List<PostList>
getNumPostsInNameSpace(String)	long
getKeysInNameSpace(String)	List<String>
getListSize(String)	long
getStringList(String, long)	List<String>
replaceStringList(String, List<String>)	long
replacePostLists(String, List<PostList>)	long
addNewPostFromSource(String, Post)	long
addNewPostsFromSource(String, List<Post>)	long
addNewDisplayPostList(String, PostList)	long
addNewHashTagPostList(String, PostList)	long
replaceHashTagPostLists(String, List<PostList>)	long
getAllPostsFromSource(String)	List<Post>
popFirstPostFromSource(String)	Optional<Post>
getAllHashtagPostLists(String)	List<PostList>
getNumHashTagPostLists(String)	long
getDisplayPostList(String, Integer)	Optional<PostList>
getHashTagPostList(String, Integer)	Optional<PostList>
getAllDisplayPostLists(String)	List<PostList>
getNumPostsInSources()	long
getSources()	List<String>
getAllHashTags()	List<String>
getTopHashTags(int)	List<String>
addTopHashtags(List<String>)	long
deleteFirstNPostsFromSourceQueue(String, Integer)	String
getMaxPostlists()	Integer
getTestNamespace()	String
getSourceNamespace()	String
getHashtagNamespace()	String
getNamespaceDelimiter()	String
getDisplayNamespace()	String

Classes that extend `AbstractDataAccess` Implement the following methods, a subset of those listed above:

```
    /**
     * Adds a new post to this data store's list of posts (end of queue) under a particular key.
     * If no key exists, a key-value pair is created and Post is the first element in the value list.
     *
     * @param keyString string denoting key in data store
     * @param post      Post object to be stored
     * @return length of list of posts under keyString after insertion of new post
     */
    abstract protected long addNewPost(String keyString, Post post);

    /**
     * Adds a series of posts to this data store's list of posts at a key, in order at the end of the queue.
     * If no key exists, a key-value pair is created and listOfPosts is stored in the value list.
     *
     * @param keyString   string denoting key in data store
     * @param listOfPosts list of posts to append to value list at key
     * @return length of list of posts under keyString after insertion of new posts
     */
    abstract protected long addNewPosts(String keyString, List<Post> listOfPosts);

    /**
     * Retrieves and removes post from the beginning of the queue at keyString in data store
     * If key does not exist, or list is empty, returns the empty optional
     *
     * @param keyString string denoting key in data store
     * @return The first element under keyString in data store, or the empty optional if not availalbe
     */
    abstract protected Optional<Post> popFirstPost(String keyString);

    /**
     * Retrieves all Posts under a particular keyString
     *
     * @param keyString string denoting key in data store
     * @return list of Posts at keyString; empty list if keyString does not exist in data store
     */
    abstract protected List<Post> getAllPosts(String keyString);

    /**
     * Deletes the first numPosts Posts under keyString. If numPosts is greater than the size of the list at keyString,
     * size elements are cleared.
     *
     * @param keyString string denoting key in data store
     * @param numPosts  number of posts to be deleted from beginning of list at keyString
     * @return string denoting status of trim operation
     */
    abstract protected String deleteFirstNPosts(String keyString, Integer numPosts); // todo: change return type

    /**
     * Adds a new postList entity to the beginning of this data store's list of postLists under a particular key.
     * If no key exists, a key-value pair is created and postList is stored at the beginning of the new value list.
     *
     * @param keyString string denoting key in data store
     * @param postList  postList entity to be entered at beginning of list under keyString
     * @return size of list at keyString after insertion of new postList
     */
    abstract protected long addNewPostList(String keyString, PostList postList);

    /**
     * Retrieves, but does not remove postList entity at specified index of the stack at keyString in data store
     * If key does not exist, or list is empty, returns the empty optional
     *
     * @param keyString string denoting key in data store
     * @param index     index of desired PostList under keyString in data store
     * @return The first element under keyString in data store, or the empty optional if not available
     */
    abstract protected Optional<PostList> getPostList(String keyString, Integer index);

    /**
     * Retrieves, but does not remove all postLists under keyString in data store
     * If key does not exist, or list is empty, returns an empty list
     *
     * @param keyString string denoting key in data store
     * @return the list of all postList entities under keyString in data store
     */
    abstract protected List<PostList> getAllPostLists(String keyString);

    /**
     * Returns the number of posts within a particular namespace within the data store. If namespace does not exist,
     * returns 0.
     *
     * @param nameSpace string corresponding to the desired namespace (proceeds namespace delimiter in any unique key)
     * @return the number of posts within all keys under nameSpace
     */
    abstract public long getNumPostsInNameSpace(String nameSpace);

    /**
```

```
     * Returns a list of all keys under a specified namespace. Returns an empty list if no keys exist under a specific
     * namespace, or if namespace does not exist in data store.
     *
     * @param nameSpace string corresponding to the desired namespace (proceeds namespace delimiter in any unique key)
     * @return A list containing all string keys under nameSpace
     */
    abstract public List<String> getKeysInNameSpace(String nameSpace);

    /**
     * Returns the size of the list at the specified key string
     *
     * @param keyString desired key string in data store
     * @return the size of the list stored at keyString
     */
    abstract protected long getListSize(String keyString);

    /**
     * Retrieves, but does not remove, the first length elements of the list of strings stored at the specified key string
     *
     * @return the list of strings stored at the specified key string
     */
    abstract protected List<String> getStringList(String keyString, long length);

    /**
     * Replaces the list of strings at keyString with the specified string list. Creates new list if one does not exist.
     *
     * @param keyString  key string in data store
     * @param stringList list of strings to replace
     * @return new length of list under keyString after insertion.
     */
    abstract protected long replaceStringList(String keyString, List<String> stringList);

    /**
     * Replaces the list of postLists at keyString with the specified list of postLists. Creates new list if one does not
     * exist.
     *
     * @param keyString keyString of postList collection
     * @param postLists list of postLists to replace existing postLists
     * @return string denoting status of trim operation
     */
    abstract protected long replacePostLists(String keyString, List<PostList> postLists);
```

Extensions of `AbstractDataAccess` provide an implementation of the data storage requirements with a specific method of storage and storage format. The `AbstractDataAccess` class then uses the public abstract methods provided by these classes, and builds on the protected abstract methods to provide the full functionality the Pique backend requires. These functionalities include, but are not limited to, namespace management, and special methods required by different Pique subsystems.

Redis Access Object

`RedisAccessObject` extends `AbstractDataAccess` to interface between the Pique backend and Redis, storing information in serialized binary format.

In Memory Access Object

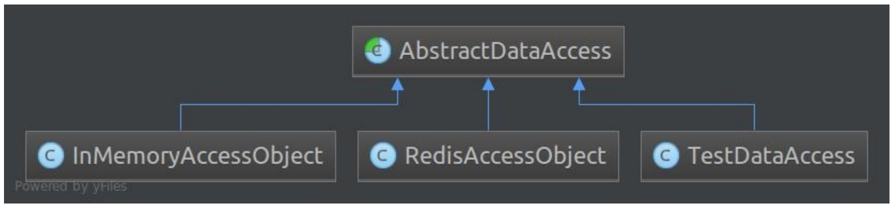
`InMemoryAccessObject` extends `AbstractDataAccess` to interface between the Pique backend and a data store maintained on memory on the same instance. This allows for quick and easy development and testing, without the need to connect to an external data store. `InMemoryAccessObject` stores and serves information using `String` keys, mapped to `Lists` of `Post` or `PostList` objects.

Test Data Access

`TestDataAccess` is a stub class used in unit tests and during front-end development so that we do not have to pull live data in order to test the front end.

Class Diagram

The following diagram shows the relationships between Data Access Classes:



Test Data Generator

Test Data Generator is included in the `dataAccess` package, providing a number of useful static methods for generating fake data to be used in unit tests and during iterative development.

Data Collection

This section details the design for our data collection component. This component is responsible for querying our set of data sources and placing the results into the data tier.

Purpose

The data collection component is a collection of threads that query sources periodically for content.

Breakdown

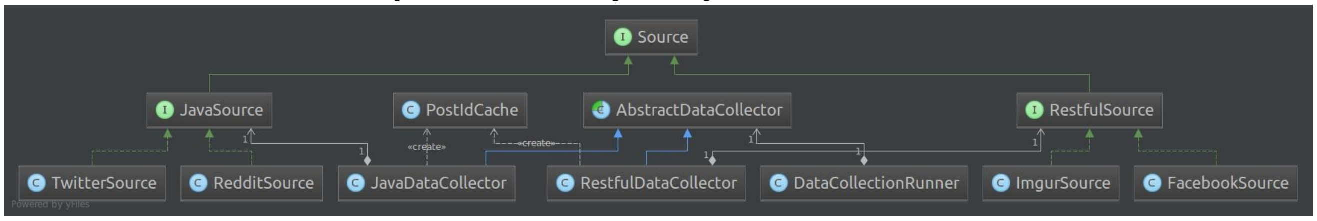
The structure of the data collection component is as follows:

- A collection thread pool is made, each given a DataCollectionRunner.
- Each runner is given an implementer of AbstractDataCollector. This is either a JavaDataCollector or a RestfulDataCollector, depending on if the source is being accessed through a java library or directly through a REST API
- Each AbstractDataCollector contains a Source object, which manages the implementation details of how to use that particular source.

Each AbstractDataCollector represents an entity that queries a particular source, and each DataCollectionRunner represents an autonomous entity that collects from that source.

Class Diagram

The data collection and source classes are represented in the following class diagram:



Responsibilities

Each DataCollectionRunner is responsible for scheduling its requests to the API such that we operate at about 75% of the API's query limit. This level of operation was chosen such that we leave some room for testing (for example through Travis CI builds) while also running the server, as they operate off of the same API key set.

Additionally, a DataCollectionRunner should notify the Sorting Node (https://github.com/pique-media/pique/wiki/Sorting-Node) when it receives data, so that the SortingNode can operate on it. This waiter-notifier architecture allows us to optimize our thread usage by letting our sorting node idle while there is nothing new to sort and our DataCollectionRunners schedule periodic requests.

Sorting Node

This section outlines the detailed design of Pique’s sorting node; the component which ultimately decides what content the user views when using Pique.

Purpose

The sorting node is a singleton module within the Pique architecture, which runs periodically to generate a new set of data which shall be displayed on the front-end.

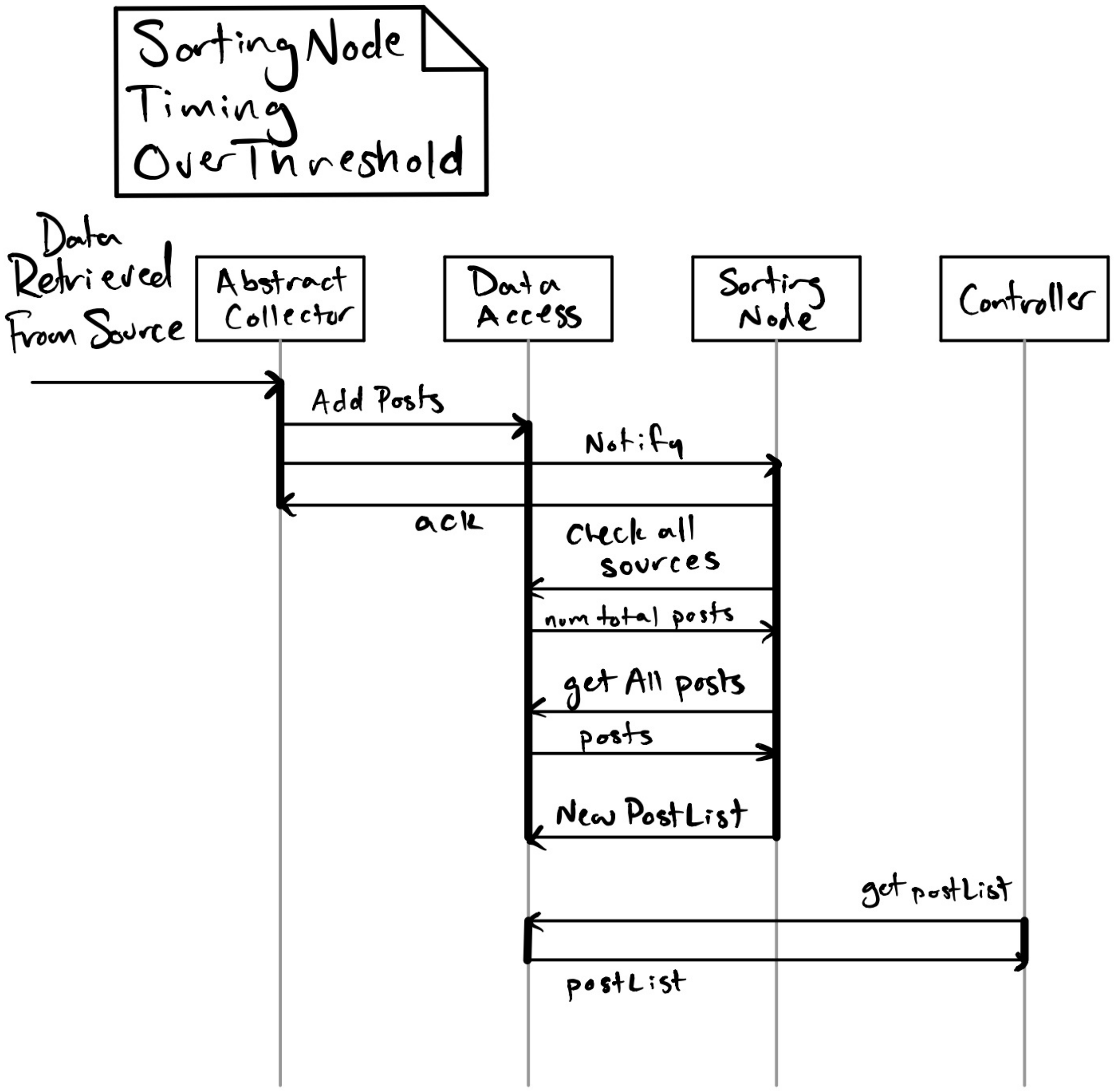
Responsibilities

The sorting node is responsible for parsing all posts gathered, creating a popularity score for each post based on the meta-data gathered, and sorting posts into particular bins based on popularity, change of popularity over time (popularity velocity), and by specific hashtags.

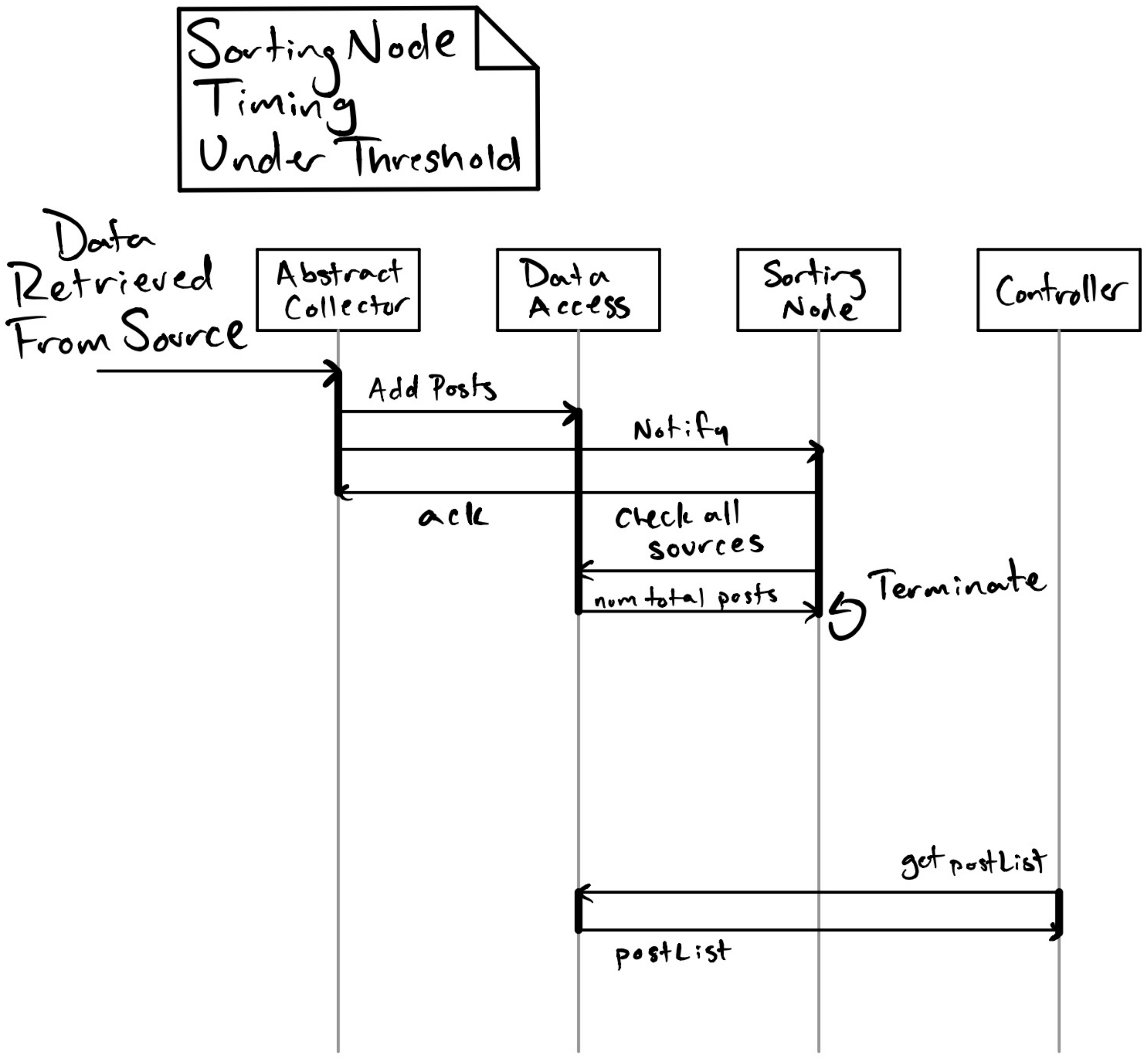
Timing

The sorting node needs to operate periodically. While this could be done on a pre-determined schedule, we may see irregularities in content production, which could result in a lack of data being displayed in some cases. Instead, we have decided to implement a “notify-and-check” system. With this system, each content gatherer will notify the sorting node when new data is loaded into the data store. The sorting node will then check on the number of posts available for processing, and decide whether to process this data based on this number.

The following timing diagram describes this process in the case that there are a sufficient number of posts available for processing:

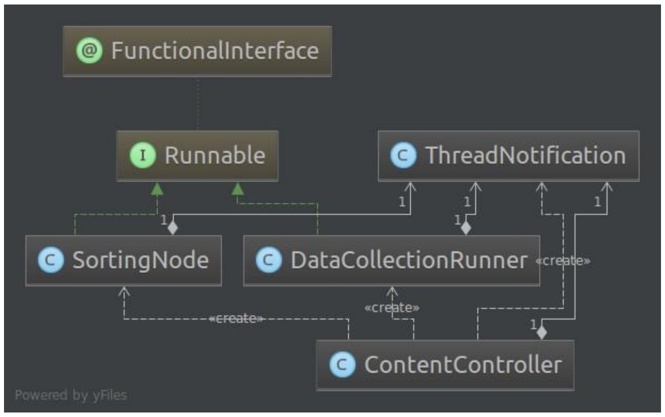


This second timing diagram describes the flow in the case that there are an insufficient number of posts available for processing:



Note that the user request initiates the interaction between a Content Controller and the Data Access class, and that this request can occur asynchronously from the rest of the process described. The data that is returned during this process is always the most recently created collection of objects to be displayed. In the case that there is no information to be displayed (sorting node has not yet successfully run), an empty data set is returned, and the Controller is then responsible for informing the user of this particular state.

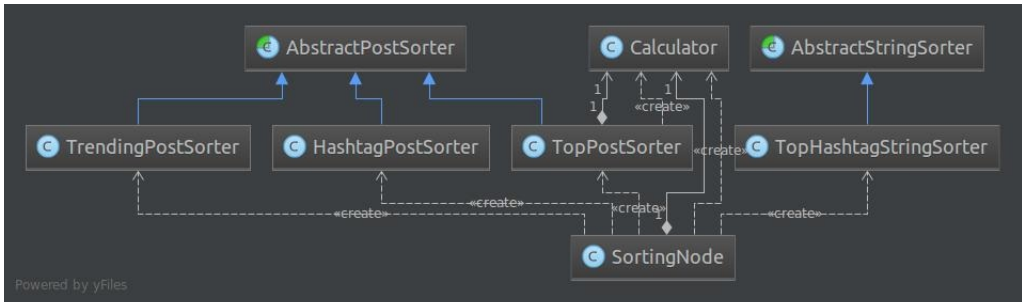
The relationships between the Sorting Node, Data Collection Runner (which controls a data collector), Data Storage, and the overarching Content Controller class is displayed in the following class diagram:



Implementation

The `SortingNode` class is an implementation of `Runnable`. Its `run` method simply waits on a notification from one of the data collectors indefinitely. Once a notification is received, it invokes its own method `sort`.

The sorting node's method `sort` first evaluates whether there are enough newly gathered posts to process (as specified by a configurable threshold), as described in the Timing section above, and exits if there are not enough posts. If there are a sufficient number of posts to proccess, the Sorting node invokes a number of `Sorter` classes, which sort various content based on specific criteria. The relationship between the Sorting Node and these Sorters is shown below.



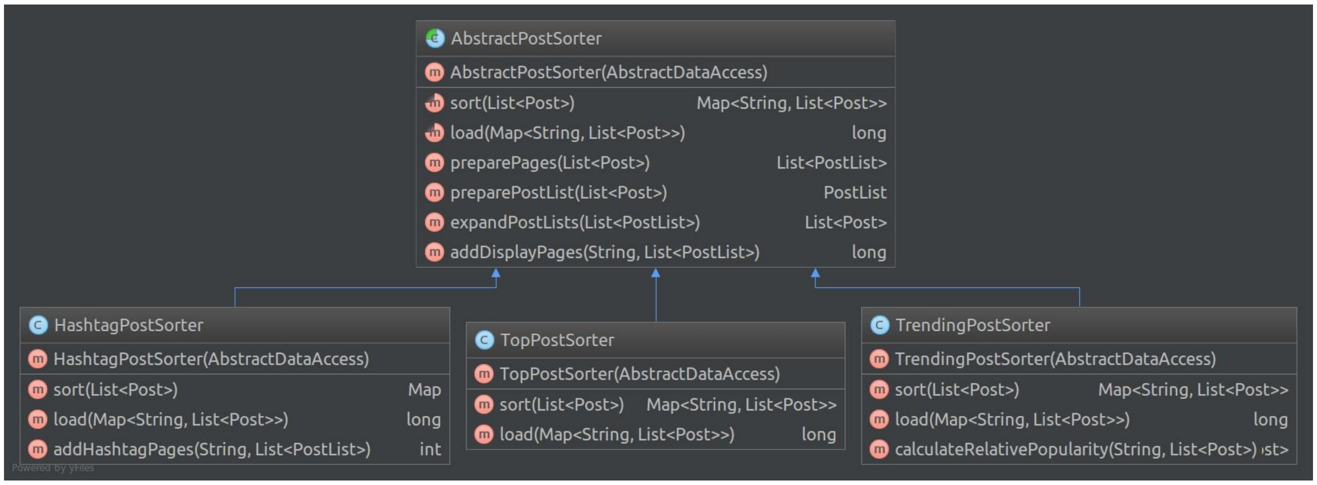
The `SortingNode` class is responsible for calculating the popularity score of each new post before running each of the different sort functionalities on the set of new posts (and in some cases, existing posts)by use of a `Calculator` object.

Sorters

There are two types of sorters, `PostSorter` and `StringSorter`. Each sorts collections of content referenced to in their name.

Post Sorter

The main sorter classes used in Pique extend `AbstractPostSorter`, which sorts a collection of Post entities. These classes support two main methods, `sort`, which sorts the specified posts according to the Class' own criteria, and `load`, which stores the sorted data in the data store for consumption by the user. The relationship between `AbstractPostSorter` and each of the individual Post Sorters follows.


AbstractPostSorter
AbstractPostSorter(AbstractDataAccess)
sort(List<Post>) Map<String, List<Post>>
load(Map<String, List<Post>>) long
preparePages(List<Post>) List<PostList>
preparePostList(List<Post>) PostList
expandPostLists(List<PostList>) List<Post>
addDisplayPages(String, List<PostList>) long
HashtagPostSorter
HashtagPostSorter(AbstractDataAccess)
sort(List<Post>) Map
load(Map<String, List<Post>>) long
addHashtagPages(String, List<PostList>) int
TopPostSorter
TopPostSorter(AbstractDataAccess)
sort(List<Post>) Map<String, List<Post>>
load(Map<String, List<Post>>) long
TrendingPostSorter
TrendingPostSorter(AbstractDataAccess)
sort(List<Post>) Map<String, List<Post>>
load(Map<String, List<Post>>) long
calculateRelativePopularity(String, List<Post>) st>
Powered by yFiles

Top Post Sorter

The `TopPostSorter` class sorts new posts and existing top posts according to their popularity score. The popularity score of old posts is recalculated during sorting to account for the age of the posts. The newly sorted posts are then loaded into the top post display channel, replacing the existing posts.

Trending Post Sorter

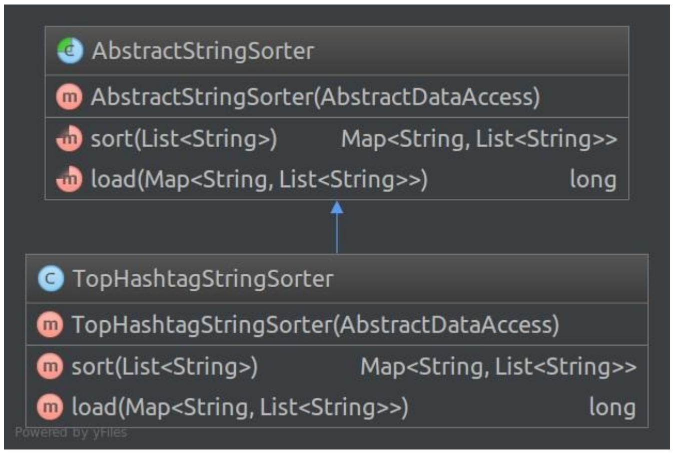
The `TrendingPostSorter` class sorts new posts based on their popularity velocity, which is determined by the difference in popularity score since the particular post was last evaluated. The newly sorted posts are then loaded into the trending display channel.

Hashtag Post Sorter

The `HashtagPostSorter` class sorts new posts based on the hashtags contained within each post. Posts are sorted into individual channels by hashtag, and loaded into the top of the particular hashtag display channel.

String Sorter

String sorters sort sets of strings. Similar to Post Sorters, all string sorters extend `AbstractStringSorter`. These classes also support similar `sort` and `load` methods, but instead sort collections of strings. The relationship between `AbstractStringSorter` and its extensions is as follows.


AbstractStringSorter
AbstractStringSorter(AbstractDataAccess)
sort(List<String>) Map<String, List<String>>
load(Map<String, List<String>>) long
TopHashtagStringSorter
TopHashtagStringSorter(AbstractDataAccess)
sort(List<String>) Map<String, List<String>>
load(Map<String, List<String>>) long
Powered by yFiles

Top Hashtag String Sorter

The `TopHashtagStringSorter` class determines the list of current top hashtags based on the number of posts stored under each hashtag channel in the data store at any given time. Its `sort` method can take a list of strings as input, but if this list of strings is `null` or an empty list, it will instead use the entire set of available hashtags currently available.

Calculator

The `Calculator` class provides methods to calculate the popularity score of posts or lists of posts. This is done using a weighted sum of likes, comments, and shares (or equivalent metrics for sources which may not fit these exact metrics), with a linear time decay. The time decay is calculated such that the popularity score of a post is scaled linearly to 0 over the course of three days in the Pique system.

For Developers

Cloning the Pique Repo

Clone the master branch (or, if you're feeling particularly adventurous, clone a feature branch) at The Pique Repository (https://github.com/pique-media/pique)
using git.

Setting up Play Framework

Visit the Play Installation page (https://www.playframework.com/documentation/2.5.x/Installing) and follow the instructions detailed there.

Downloading Activator

Activator is a very useful build tool, based on the sbt package manager. Download Activator (https://downloads.typesafe.com/typesafe-activator/1.3.12/typesafe-activator-1.3.12.zip), unzip, and add it
to your `PATH` .

Using Activator

If you haven't already, clone the Pique source code (https://github.com/edolinsky/pique), and set up your environment variables, as described in the **Environment Variables** section of this document.

Within the `pique` directory in your terminal, you can run a local instance of Pique by issuing
`activator run`
Pique will then start up on `localhost:9000` . You may need to wait a few moments for the Sorter to run for the first time before content is available on the webpage.

You can run unit tests by issuing
`activator test`

You can create a production build of Pique for use on an Amazon Web Services Elastic Beanstalk application by running
`activator dist`
The `.zip` package will then be available in the `pique/target/universal/` directory. See Deploying a Pique Application (https://www.playframework.com/documentation/2.5.x/Deploying)
for more information.

Testing in an IDE

If your IDE supports the Play framework or sbt, you should be able to import the Pique project (https://www.playframework.com/documentation/2.5.x/IDE#Setup-sbteclipse) accordingly. You should then be able to run the unit test suite, provided you have imported the required environment variables

Environment Variables

Environment variables are used for configuring data storage, external API access for data sources, and other things we need to configure, but don't necessarily want to display on our public repository. If you need them (for grading of this project, for example) and we haven't supplied them, please ask!

Application Environment Variables

The following environment variables are required by the Pique application itself:

- `runtime_env` - setting this to `production` will result in Pique using Redis as a data store. Any other field (or none) will result in Pique using its own 'in memory' data store.

Content Collector Environment Variables

The following environment variables are required by Pique's content collector classes

Twitter Collector Environment Variables

- `twitter4j_consumerKey` - Twitter API consumer key
- `twitter4j_consumerSecret` - Twitter API consumer secret
- `twitter4j_accessToken` - Twitter API access token
- `twitter4j_accessTokenSecret` - Twitter API access secret

Reddit Collector Environment Variables

- `reddit_user` - Reddit username
- `reddit_pass` - Reddit password
- `reddit_client_id` - Reddit client ID
- `reddit_secret` - Reddit client secret

Imgur Collector Environment Variables

- `imgur_client_id` - Imgur application client ID
- `imgur_client_secret` - Imgur application client secret

Data Access Environment Variables

When using Redis, the following variables are required:

- `redis_url` - URL of Redis instance
- `redis_port` - Port at `redis_url` on which Redis is listening

Sorting Node Environment Variables

The sorting node requires the following environment variables to be set:

- `sorting_threshold` - number of posts required to be stored in source channels before the sorting node runs
- `posts_per_page` - number of posts to be stored in each display page

Production Environment Variables

When running Pique in a production environment (Elastic Beanstalk), the following variables are required:

- `http_port` - http port on which the load balancer listens
- `app_secret` - Secret string (can be any string, but needs to be kept private as the application is not secure if public)

Test Environment Variables

When *testing* Redis, the following variables are required:

- `data_source` - set to `redis` to enable Redis tests to run

Deploying Pique

Once you have the Activator tool set up, you should be able to create a deployable package using `activator dist` in your command line. This will create a zip package in the `pique/target/universal/` directory that you can upload to an AWS ElasticBeanstalk application. This requires environment variables, as detailed in the **Environment Variables** section.

Structure

Pique directory structure mainly follows the Anatomy of a Play! Application (https://www.playframework.com/documentation/2.5.x/Anatomy). The application itself is located in `app`, which contains `controllers`, `models`, `filters`, and `services`, where most of the backend components are implemented, as well as Scala-templated HTML for display on the browser. `javascripts`, `stylesheets`, and `images` passed to the browser are located in `public`. Other directories of interest are `test`, where unit tests live, `conf`, where configuration files are stored (including `routes.conf` in which the HTTP endpoints are routed to specific controllers), and `logs`.

Design Patterns

Pique takes most of its design from the Model-View-Controller design pattern, but also makes use of singletons, templates, builders, observers, data access objects, and possibly some others we may have forgotten about.

Design Changes and Rationale

High Level Design

- *Sun. Oct. 20*: Validation Section added
- *Mon. Oct. 28*: Notion of PostList Entities added to aid in protobuf deserialization on the front-end. (Protobuf serialization later switched to JSON, but PostList Entities will still be used as a paging mechanism for the front end.)
- *Tue. Nov 1*: Reference to workflow specification in Test Plan Document
- *Wed. Nov. 9*: Added updated class diagram

Data Tier

- *Wed. Oct. 26*: The original Data Tier Design & Implementation relied too heavily on the functionalities of Redis,and the specific data storage format of serialized `Post` / `Postlist` objects to Binary. Specific functionalities were changed so that the Pique backend deals only in mappings of String keys to Lists of `Post` / `Postlist` objects. This was designed such that should we change the format of data storage from binary (for example, possibly to JSON strings), or storage service (as in the case of InMemory vs Redis), one class that extends `AbstractDataAccess` need only be implemented.
- *Thu. Oct. 27*: Notion of a `serializer` added to serve data to the front end in different formats, and allow us to easily change the format of this data.
- *Thu. Nov. 3*: `getNumPostListsInNameSpace` , `getKeysInNameSpace` , `deleteFirstNPosts` , and `getAllPostLists` Methods added to better serve functionality required by the sorting node, as per Sorting Node Design (https://github.com/pique-media/pique/wiki/Sorting-Node).
- *Mon. Nov. 7*: PostList expiry in Redis clarified.
- *Wed. Nov. 9*: Change to JSON Serialization from Protobuf binary wire format; implementation of deserialization has proven to be superfluous for our purposes, and JSON is much easier to work with on the front-end. To aid in this, we chose to use Google's gson to serialize PostList objects. Changed `AbstractDataAccess.popFirstPostList` to `AbstractDataAccess.getPostList` to better serve front-end paging, moved namespace management responsibilities to `AbstractDataAccess` , and refactored this class to serve namespaces.

Sorting Node

- *Mon. Nov. 3*: In order to save the sorting algorithm extra trouble of filtering posts that do not fit the descriptionof what Pique should display to the user (not top content, not trending), a notion of a pre-filter within the data collection classes has been added.
- *Tue. Nov. 8*: String sorter added so that we can support sorting top hashtags.
- *Tue. Nov. 15*: Sorting Node designed to include Sorter abstract classes, to provide better extensibility and ease of implementation of new functionalities in the future