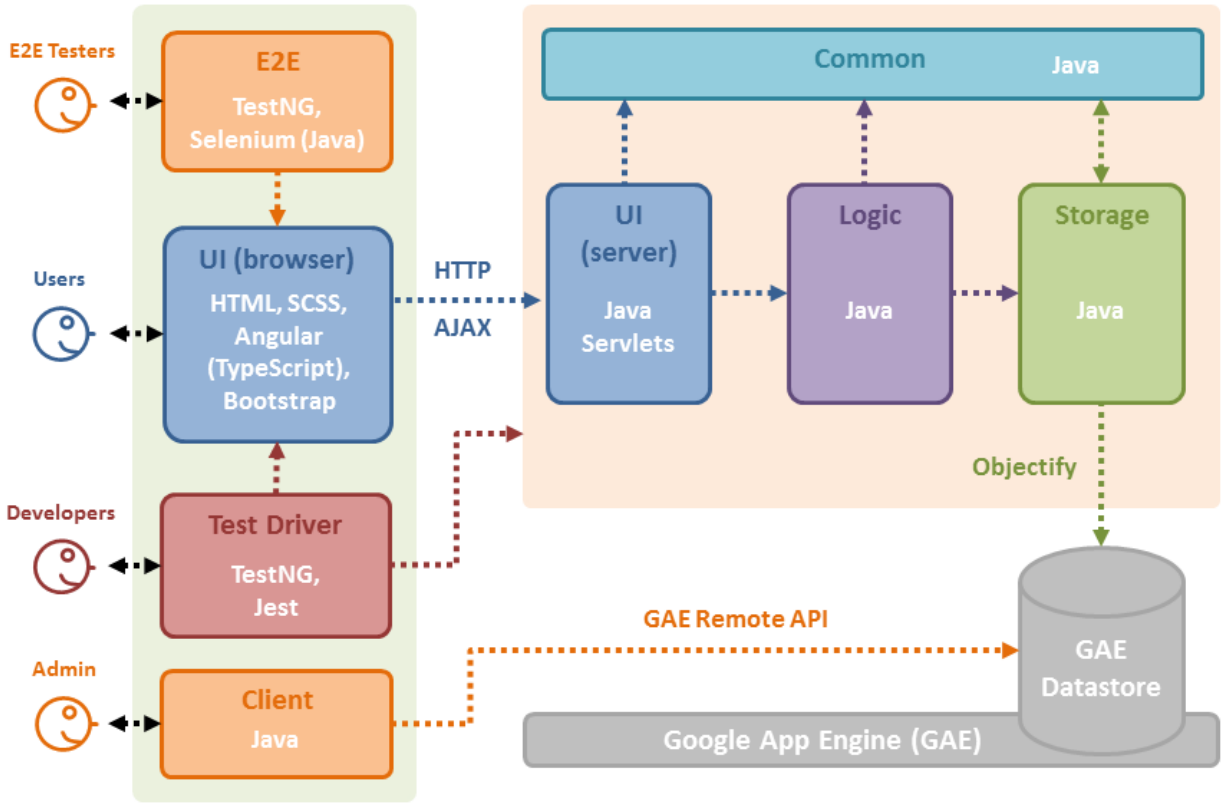
Design

- Architecture
- UI Component
- Logic Component
- Storage Component
- Common Component
- Test Driver Component
- E2E Component
- Client Component

Architecture

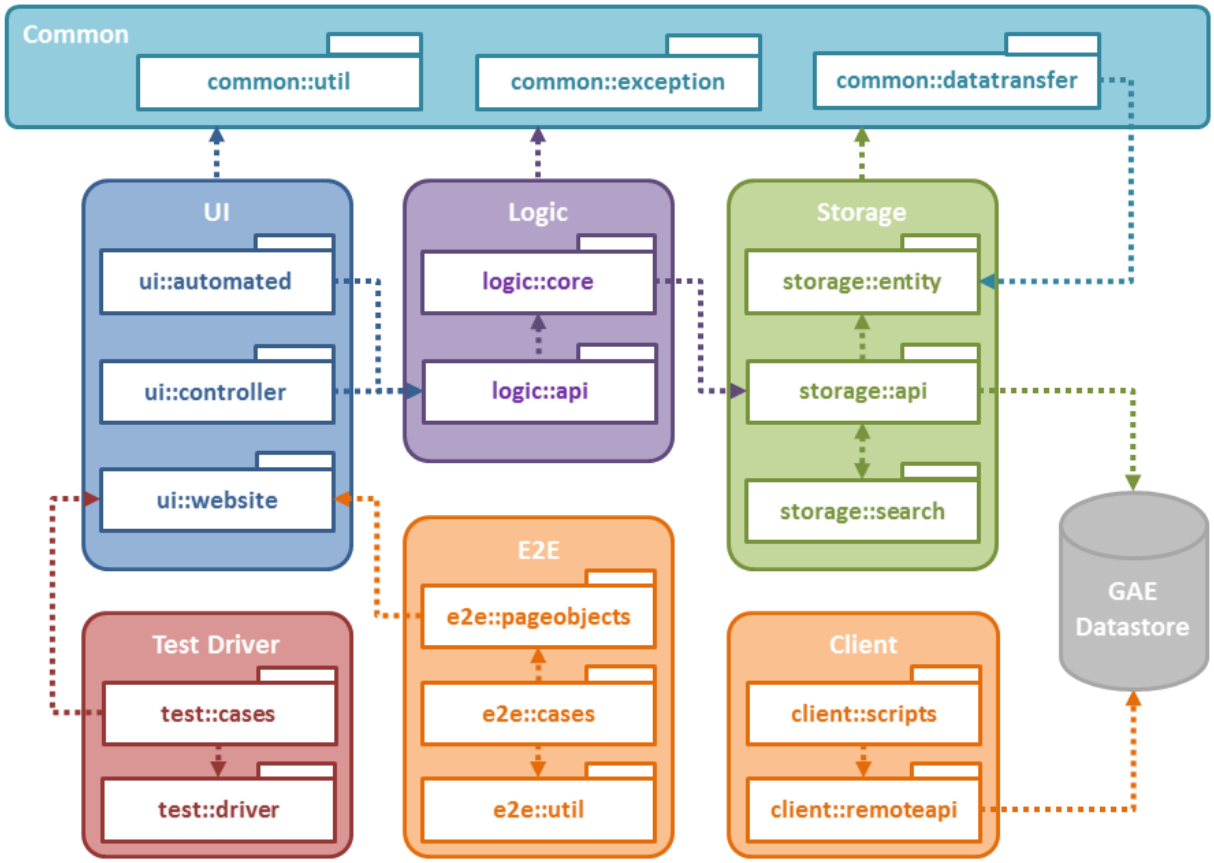


TEAMMATES is a Web application that runs on Google App Engine (GAE). Given above is an overview of the main components.

- **UI (Browser)**: The UI seen by users consists of Web pages containing HTML, CSS (for styling) and JavaScript (for client-side interactions such as sorting, input validation, etc.). This UI is a single HTML page generated by Angular framework. The initial page request is sent to the server over HTTP, and requests for data are sent asynchronously with AJAX.
- **UI (Server)**: The entry point for the application back end logic is designed as a REST-ful controller.
- **Logic**: The main logic of the application is in POJOs (Plain Old Java Objects).
- **Storage**: The storage layer of the application uses the persistence framework provided by **GAE Datastore**, a NoSQL database.
- **Test Driver**: TEAMMATES makes heavy use of automated regression testing. Test data is transmitted using JSON format.
  - `TestNG` is used for Java testing (all levels) and `Jest` for JavaScript unit-testing.
  - `HttpUnit` is used to set up a simulated web server in servlet-level tests, where an actual web server is not required.
- **E2E**: The E2E (end-to-end) component is used to interact with the application as a whole with Web browsers. Its primary function is for E2E tests.
  - `Selenium (Java)` is used to automate E2E testing with actual Web browsers.

- **Client**: The Client component can connect to the back end directly without using a Web browser. It is used for administrative purposes, e.g. migrating data to a new schema.
- **Common**: The Common component contains utility code (data transfer objects, helper classes, etc.) used across the application.

The diagram below shows how the code in each component is organized into packages and the dependencies between them.

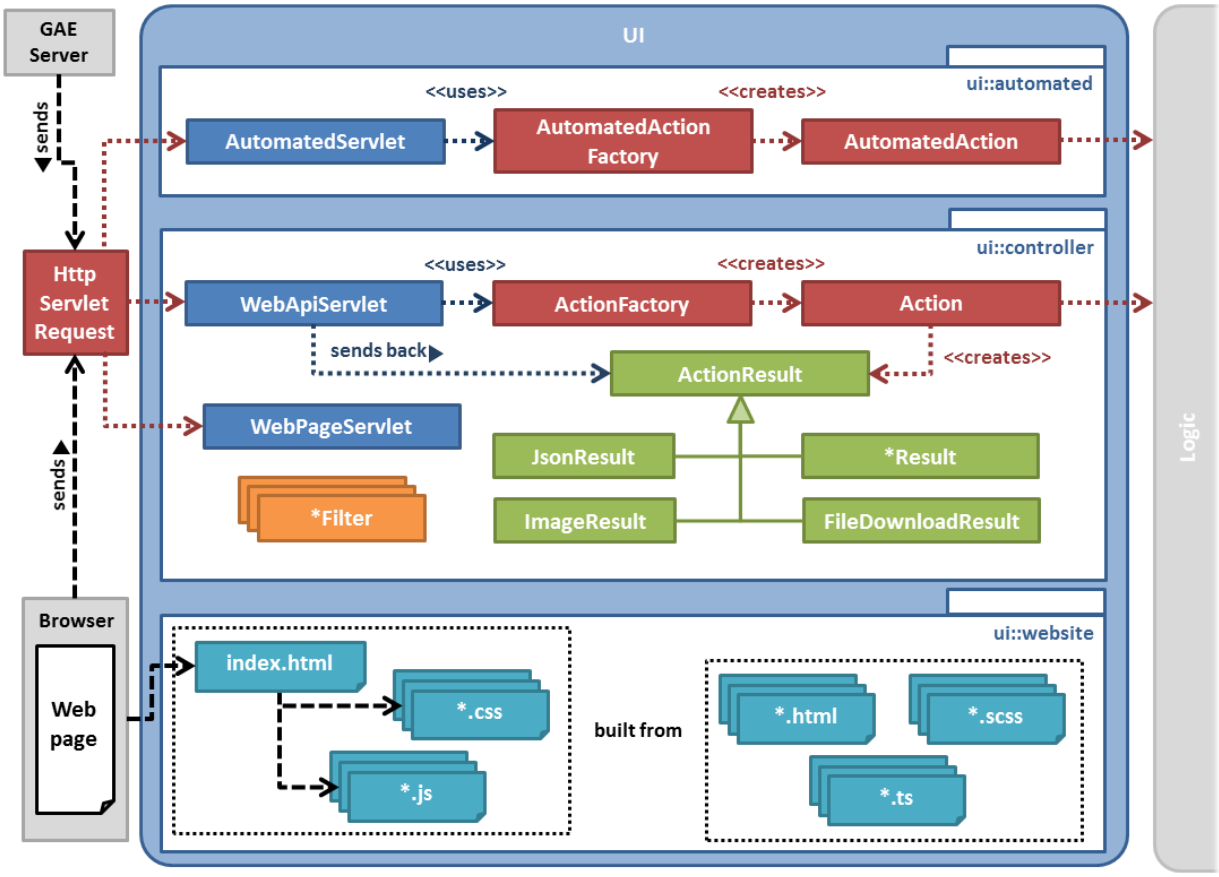


Notes:

- `[logic] - [ui::website] - [ui::controller]` represent an application of `Model-View-Controller` pattern.
  - `ui::website` is not a real package; it is a conceptual package representing the front-end of the application.

UI Component

The diagram below shows the object structure of the UI component.



Notes:

- `ui::website` is not a Java package. It is written in Angular framework and consists of HTML, SCSS, and TypeScript files. The framework will build those files into HTML, CSS and JavaScript files ready to be used by standard Web browsers.

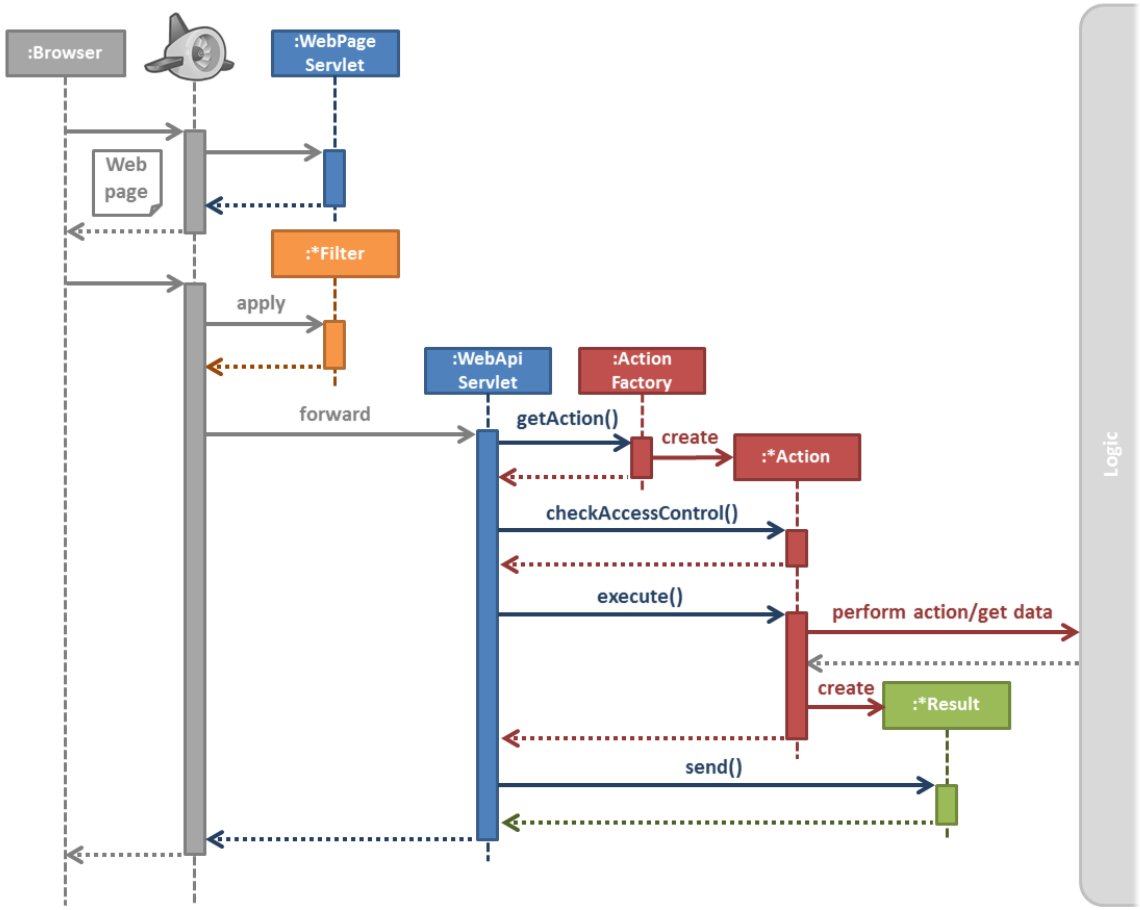
The UI component is the first stop for 99% of all requests that are received by the application. Such a request will go through the following steps:

1. Request received by the GAE server.
2. Custom filters are applied according to the order specified in `web.xml`, e.g. `OriginCheckFilter`.
3. Request forwarded to a `*Servlet` object as specified in `web.xml` for further processing, depending on the type of request.

There are two general types of requests: user-invoked requests and automated (GAE server-invoked) requests, which are processed differently.

User-invoked requests

User-invoked requests are all requests made by the users of the application, typically from the Web browser (i.e. by navigating to a particular URL of the application). The request will be processed as follows:



The initial request for the web page will be processed as follows:

1. Request forwarded to `WebPageServlet`.
2. `WebPageServlet` returns the built single web page (`index.html`).
3. The browser will render the page and execute the page scripts, most of the time requiring AJAX requests to the server.

Subsequent AJAX requests sent to the server will be processed as follows:

1. Request forwarded to the `WebApiServlet`.
2. `WebApiServlet` uses the `ActionFactory` to generate the matching `Action` object, e.g. `InstructorHomePageAction`.
3. `WebApiServlet` executes the action.
    i. The `Action` object checks the access rights of the user. If the action is allowed, it will be performed, interacting with the `Logic` component as necessary.
    ii. The `Action` packages and processes the result into an `ActionResult` object. The most common format is `JsonResult` (requests for obtaining data or processing existing data), and other formats are defined as necessary, e.g. `FileDownloadResult` (e.g. downloading feedback session report) and `ImageResult` (e.g. profile pictures).
4. `WebApiServlet` sends the result back to the browser which will then process it on the front-end.

Requests for static asset files (e.g. CSS, JS files, images) are served directly without going through `web.xml` configuration at all.

The Web API is protected by two layers of access control check:

- Origin check: This mitigates CSRF attack.
- Authentication and authorization check: This checks if the logged in user (or lack thereof) has sufficient privileges to trigger the API's actions.

Special keys (`csrf key` and `backdoor key`) can be used to bypass each of the checks, typically for testing purpose. Those keys are strings known only to the person who deployed the application (typically, the administrator).

Automated requests

Automated requests are all requests sent automatically by the GAE server during specific periods of time. This type of request will be processed as follows:

1. The source of the request will be checked for administrator privilege. If this privilege is absent (e.g. non-administrator users trying to invoke the automated actions), the request will be dropped and a `403 Forbidden` status will be returned.
    - Requests generated by the GAE server are equipped with this privilege.
    - Administrators can manually invoke these requests; this is particularly useful in testing the actions associated with those requests.
2. Request forwarded to the `AutomatedServlet` .
3. `AutomatedServlet` uses the `AutomatedActionFactory` to generate the matching `AutomatedAction` object, e.g. `CompileLogsAction` .
4. `AutomatedServlet` executes the action.
5. The corresponding `AutomatedAction` will be performed, interacting with the `Logic` component as necessary.

GAE server sends such automated requests through two different configurations:

- Cron jobs: These are jobs that are automatically scheduled for a specified period of time, e.g. scheduling feedback session opening reminders. It is configured in `cron.xml` .
- Task queue workers: These are hybrids of user-invoked and GAE-invoked in that they are queued by users (i.e. users request for the tasks to be added to queue), but executed by GAE (i.e. GAE determines when and which tasks in the queue are executed at any point of time). This is typically used for tasks that may take a long time to finish and can exceed the 1 minute standard request processing limit imposed by GAE. It is configured in `queue.xml` as well as the `TaskQueue` nested class of the Const class.

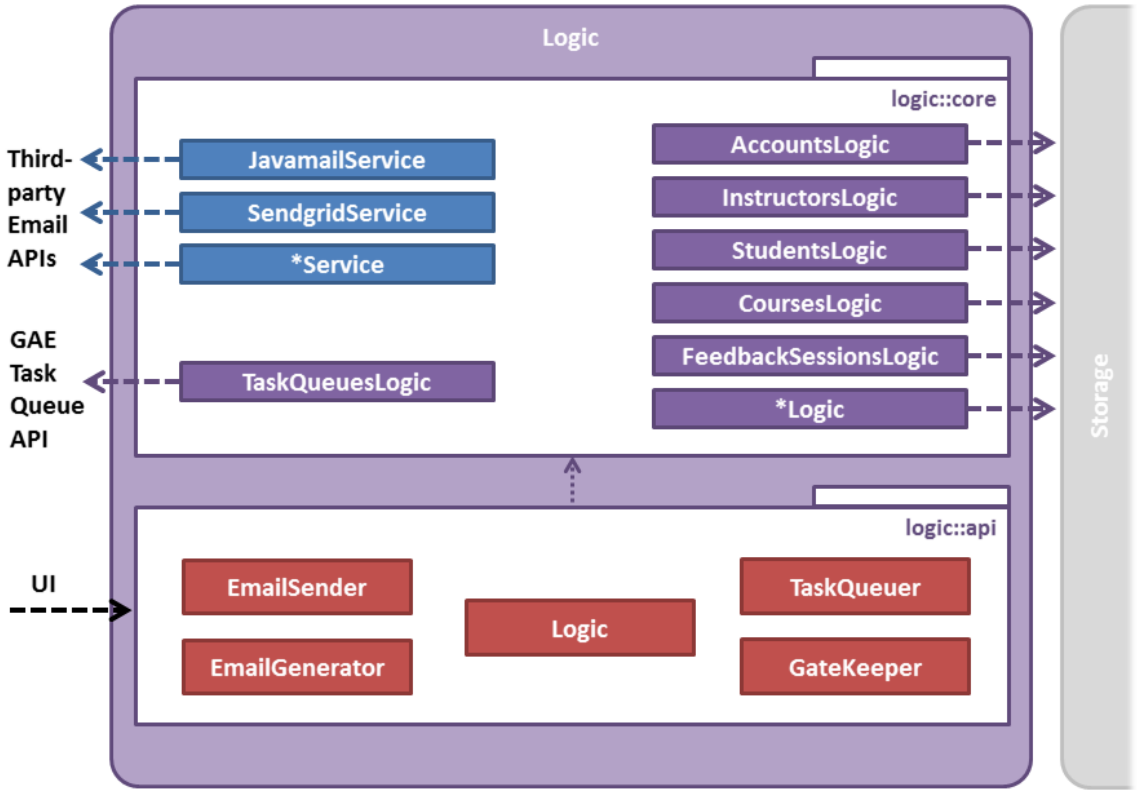
Template Method pattern

- Since the high-level workflow of processing a request is same for any request (differing by the two request types only), we use the Template Method pattern to abstract the process flow into the `Action` and `AutomatedAction` classes.

Logic Component

The `Logic` component handles the business logic of TEAMMATES. In particular, it is responsible for:

- Managing relationships between entities, e.g. cascade logic for create/update/delete.
- Managing transactions, e.g. ensuring atomicity of a transaction.
- Sanitizing input values received from the UI component.
- Providing a mechanism for checking access control rights.
- Connecting to GAE-provided or third-party APIs, e.g. for adding tasks to the task queue and for sending emails with third-party providers.



Package overview:

- **`logic.api`** : Provides the API of the component to be accessed by the UI.
- **`logic.core`** : Contains the core logic of the system.

Logic API

Represented by these classes:

- `Logic` : A Facade class which connects to the several `*Logic` classes to handle the logic related to various types of data and to access data from the `Storage` component.
- `GateKeeper` : Checks access rights of a user for a given action.
- `EmailGenerator` : Generates emails to be sent.
- `EmailSender` : Sends email with the provider chosen based on the build configuration. It connects to the email provider by using the appropriate `*Service` class.
- `TaskQueuer` : Adds tasks to the task queue. It connects to GAE's task queue API.

Policies

Access control:

- Although this component provides methods to perform access control, the API itself is not access controlled. The UI is expected to check access control (using `GateKeeper` class) before calling a method in the `Logic` .

API for creating entities:

- Null parameters: Causes an assertion failure.
- Invalid parameters: Throws `InvalidParametersException` .
- Entity already exists: Throws `EntityAlreadyExistsException` (escalated from Storage level).

API for retrieving entities:

- Attempting to retrieve objects using `null` parameters: Causes an assertion failure.
- Entity not found:
  - Returns `null` if the target entity not found. This way, read operations can be used easily for checking the existence of an entity.

API for updating entities:

- Update is done using `*UpdateOptions` inside every `*Attributes` . The `UpdateOptions` will specify what is used to identify the entity to update and what will be updated.
- Entity not found: Throws `EntityDoesNotExistException` .
- Invalid parameters: Throws `InvalidParametersException` .

API for deleting entities:

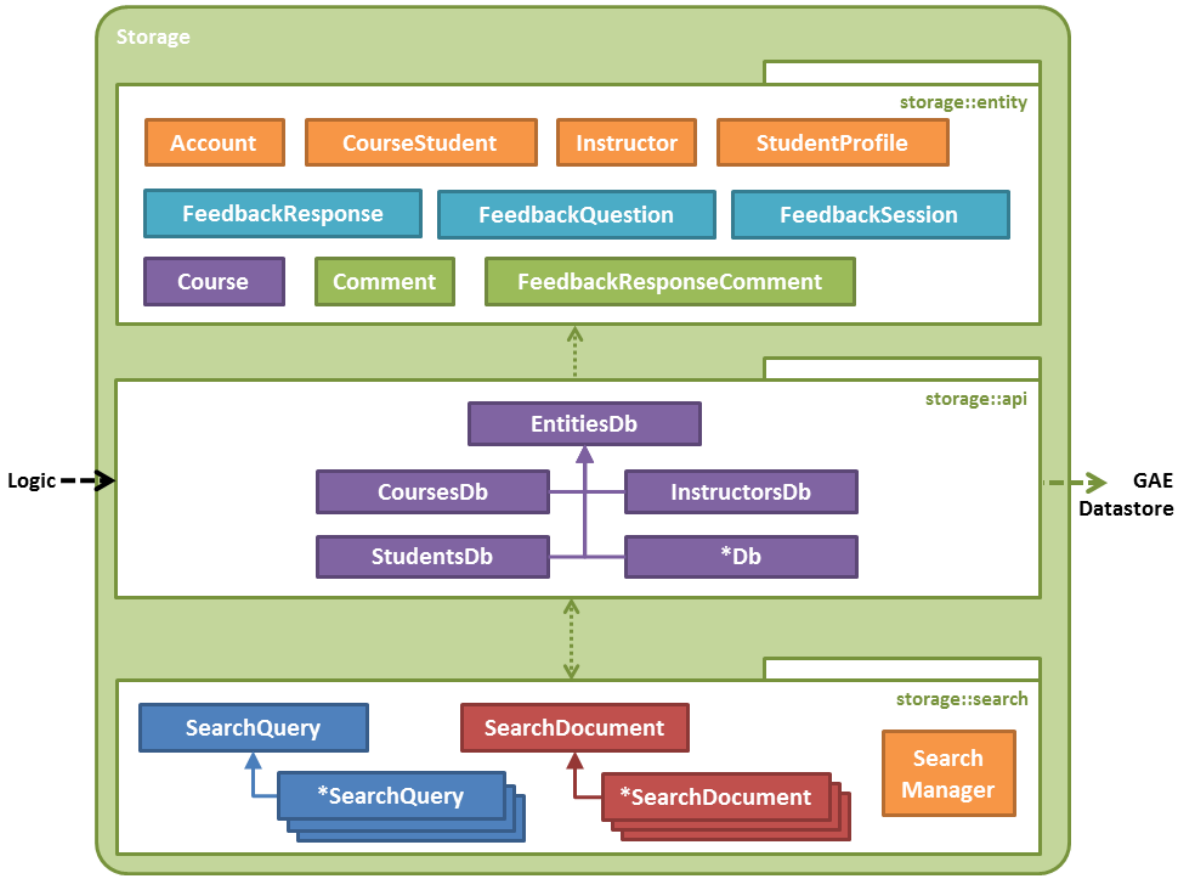
- FailDeleteSilentlyPolicy: In general, delete operation do not throw exceptions if the target entity does not exist. This is because if it does not exist, it is as good as deleted.
- Cascade policy: When a parent entity is deleted, entities that have referential integrity with the deleted entity should also be deleted. Refer to the API for the cascade logic.

Storage Component

The `Storage` component performs CRUD (Create, Read, Update, Delete) operations on data entities individually. It contains minimal logic beyond what is directly relevant to CRUD operations. In particular, it is reponsible for:

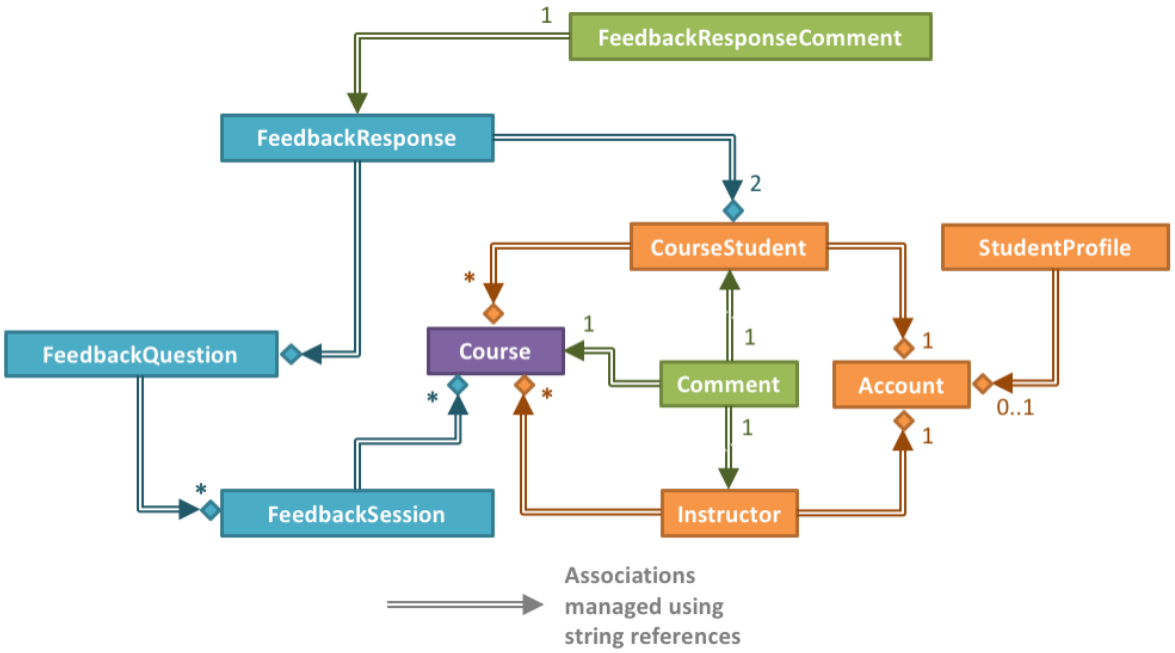
- Validating data inside entities before creating/updating them, to ensure they are in a valid state.
- Hiding the complexities of datastore from the `Logic` component. All GQL queries are to be contained inside the `Storage` component.
- Hiding the persistable objects: Classes in the `storage::entity` package are not visible outside this component to hide information specific to data persistence.
  - Instead, a corresponding non-persistent data transfer object named `*Attributes` (e.g., `CourseAttributes` is the data transfer object for `Course` entities) object is returned. These datatransfer classes are in `common::datatransfer` package, to be explained later.

The `Storage` component does not perform any cascade delete/create operations. Cascade logic is handled by the `Logic` component.



Package overview:

- **`storage.api`** : Provides the API of the component to be accessed by the logic component.
- **`storage.entity`** : Classes that represent persistable entities.
- **`storage.search`** : Classes for dealing with searching and indexing.



Note that the navigability of the association links between entity objects appear to be in the reverse direction of what we see in a normal OOP design. This is because we want to keep the data schema flexible so that new entity types can be added later with minimal modifications to existing elements.

Storage API

Represented by the `*Db` classes. These classes act as the bridge to the GAE Datastore.

Policies

Add and Delete operations try to wait until data is persisted in the datastore before returning. This is not enough to compensate for eventual consistency involving multiple servers in the GAE production enviornment. However, it is expected to avoid test failures caused by eventual consistency in dev server and reduce such problems in the live server. Note: 'Eventual consistency' here means it takes some time for a database operation to propagate across all serves of the Google's distributed datastore. As a result, the data may be in an inconsistent states for short periods of time although things should become consistent 'eventually'. For example, an object we deleted may appear to still exist for a short while.

Implementation of Transaction Control has been minimized due to limitations of GAE environment and the nature of our data schema.

API for creating:

- Attempt to create an entity that already exists: Throws `EntityAlreadyExistsException`.
- Attempt to create an entity with invalid data: Throws `InvalidParametersException`.

API for retrieving:

- Attempt to retrieve an entity that does not exist: Returns `null`.

API for updating:

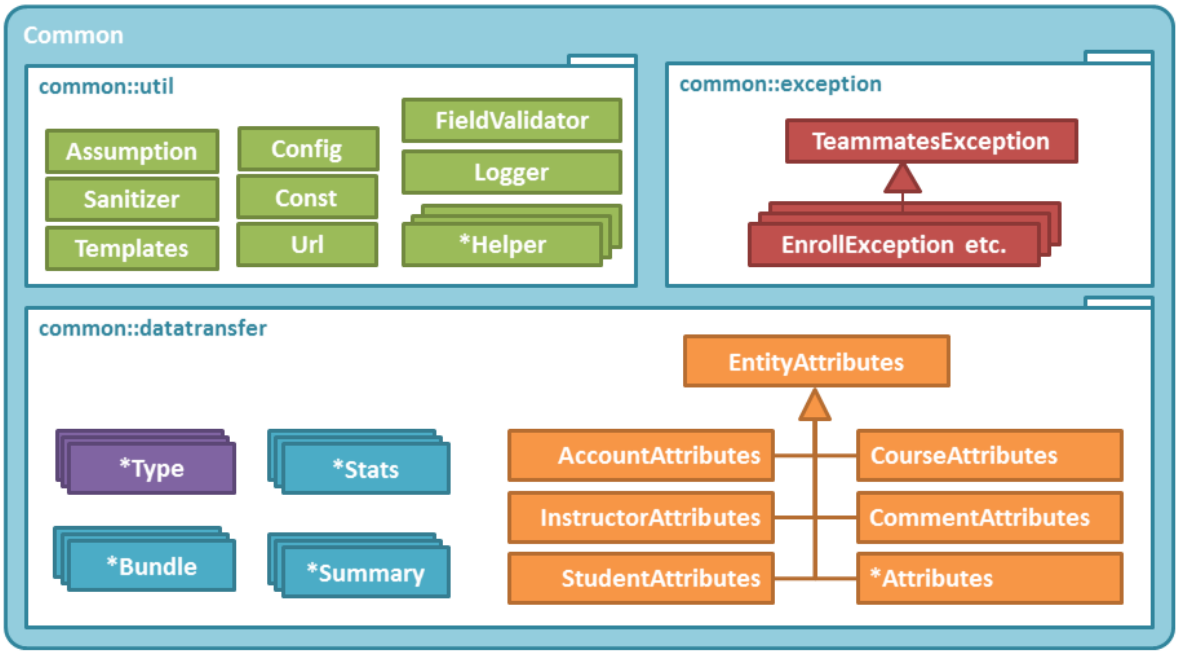
- Attempt to update an entity that does not exist: Throws `EntityDoesNotExistException`.
- Attempt to update an entity with invalid data: Throws `InvalidParametersException`.

API for deleting:

- Attempt to delete an entity that does not exist: Fails silently.

Common Component

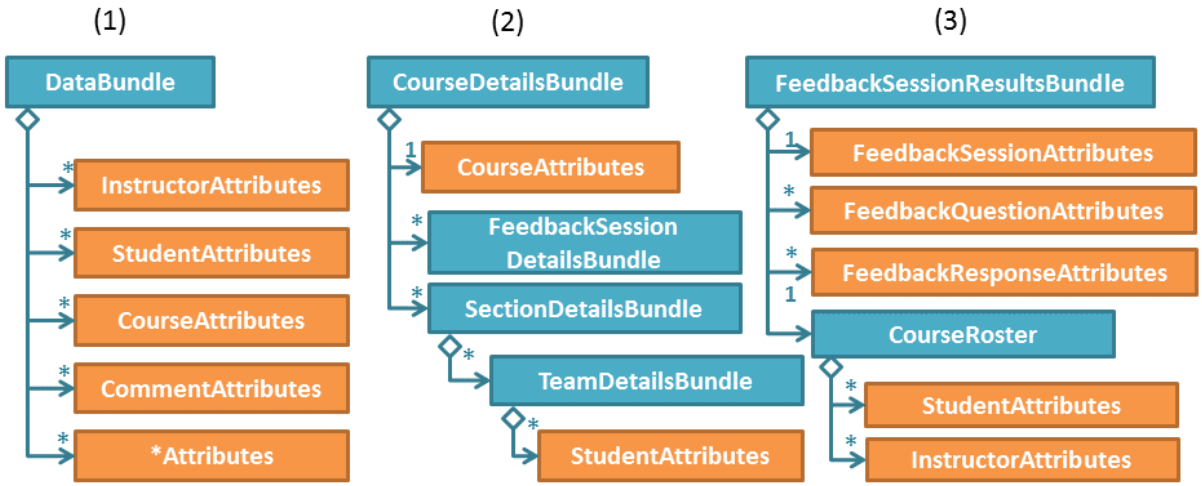
The Common component contains common utilities used across TEAMMATES.



Package overview:

- **`common.util`** : Contains utility classes.
- **`common.exceptions`** : Contains custom exceptions.
- **`common.datatransfer`** : Contains data transfer objects.

`common.datatransfer` package contains lightweight "data transfer object" classes for transferring data among components. They can be combined in various ways to transfer structured data between components. Given below are three examples.



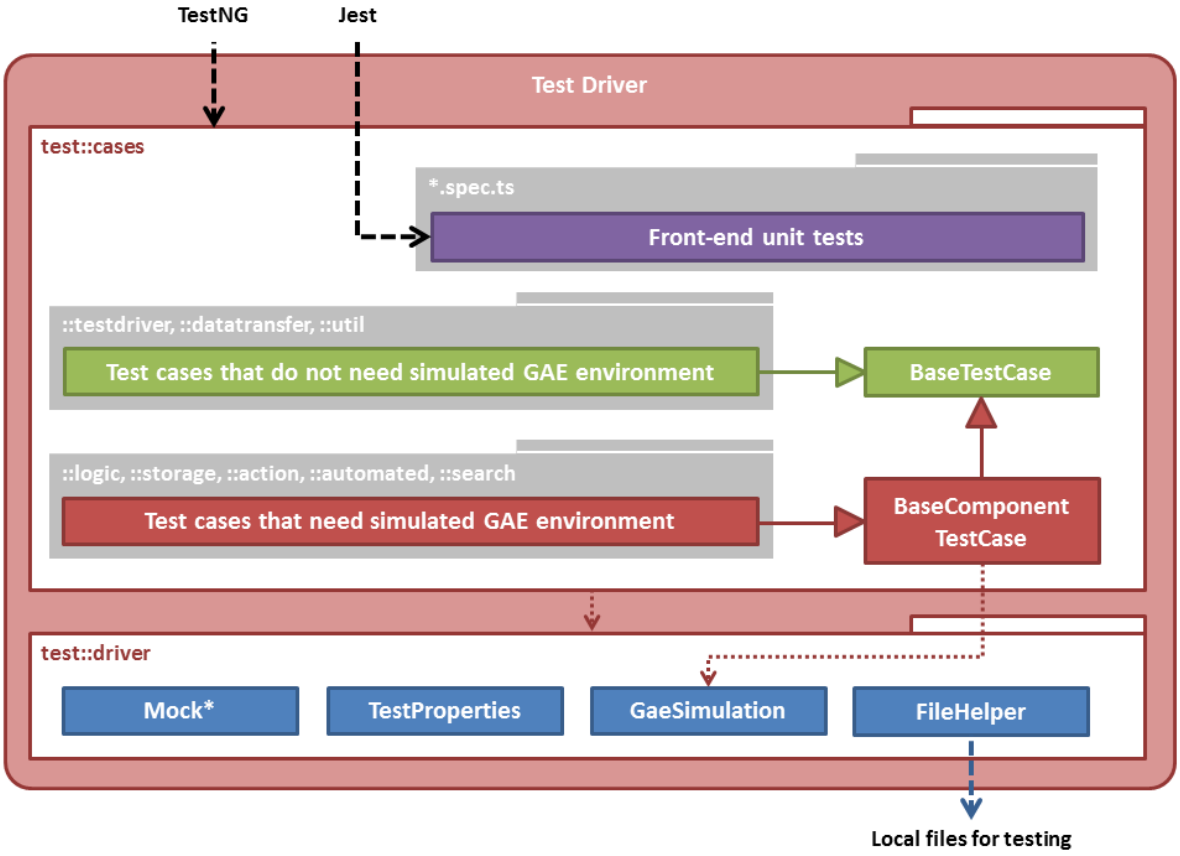
1. `Test Driver` can use the `DataBundle` in this manner to send an arbitrary number of objects to be persisted in the database.
2. This structure can be used to transfer data of a course (e.g., when constructing the home page for an instructor).

3. This structure can be used to send results of a feedback session (e.g., when showing a feedback session report to an instructor).

Some of these classes are methodless (and thus more of a data structure rather than a class); these classes use public variables for data for easy access.

Test Driver Component

This component automates the testing of TEAMMATES.



Package overview:

- **`test.driver`** : Contains infrastructure and helpers needed for running the tests.
- **`test.cases`** : Contains test cases. Sub-packages:
  - **`.testdriver`** : Component test cases for testing the test driver infrastructure and helpers.
  - **`.datatransfer`** : Component test cases for testing the datatransfer objects from the `Common` component.
  - **`.util`** : Component test cases for testing the utility classes from the `Common` component.
  - **`.logic`** : Component test cases for testing the `Logic` component.
  - **`.storage`** : Component test cases for testing the `Storage` component.
  - **`.search`** : Component test cases for testing the search functions.

- **`.webapi`** : System test cases for testing the user-invoked actions.
- **`.automated`** : System test cases for testing the system-automated actions (manually invoked during testing).

Notes:

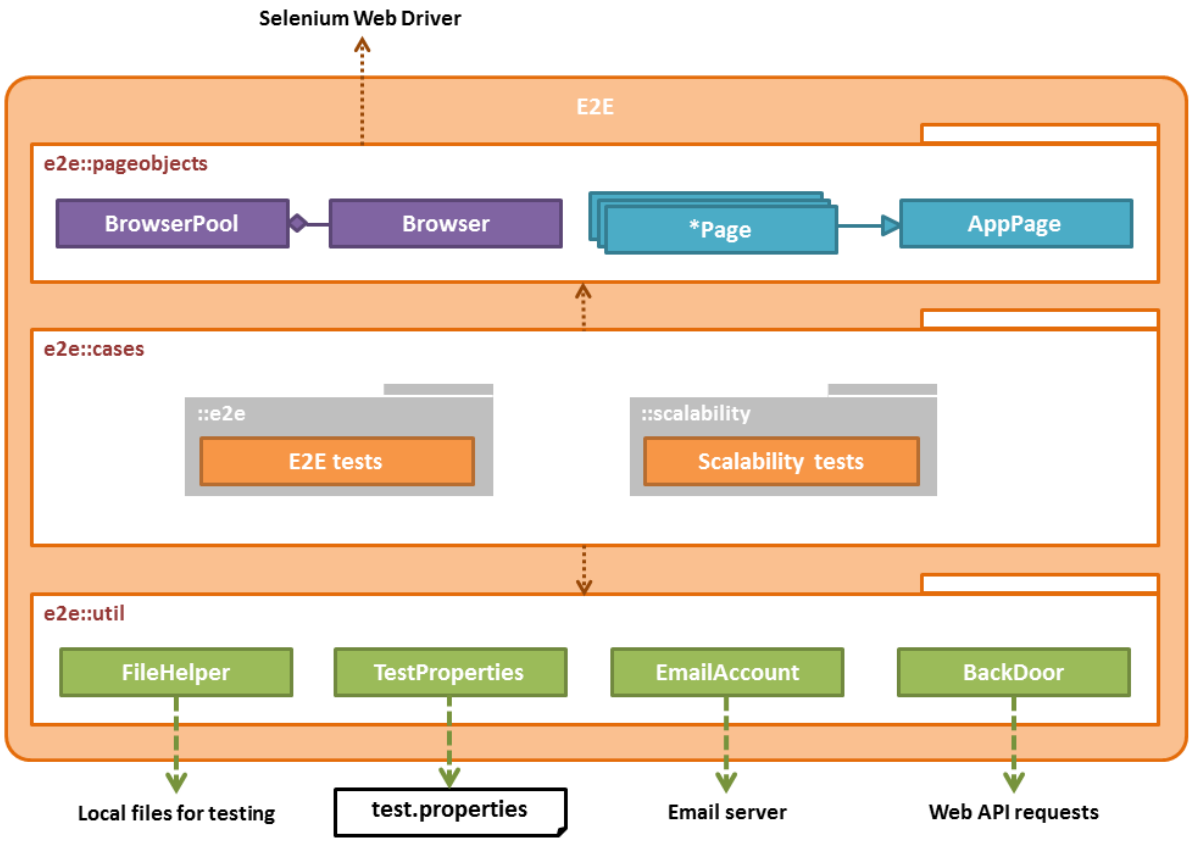
- Component tests: Some of these are pure unit tests (i.e. testing one component in isolation) while others are integration tests that test units as well as integration of units with each other.
- Front-end files (particularly TypeScript) are tested separately with `Jest`. The test cases are found in `*.spec.ts` files.

This is how TEAMMATES testing maps to standard types of testing.

```
Normal
|-----acceptance tests-----|-------------------system tests-------------------
|-----integration tests-----|-----unit tests-----|
|--------manual testing--------|------automated E2E tests------|--------------
-----automated component tests-------------------|
TEAMMATES
```

E2E Component

The E2E component has no knowledge of the internal workings of the application and can only interact either with Web browser (as a whole application) or REST API calls (for the back-end logic). Its primary function is for E2E tests and L&P (Load & Performance) tests.

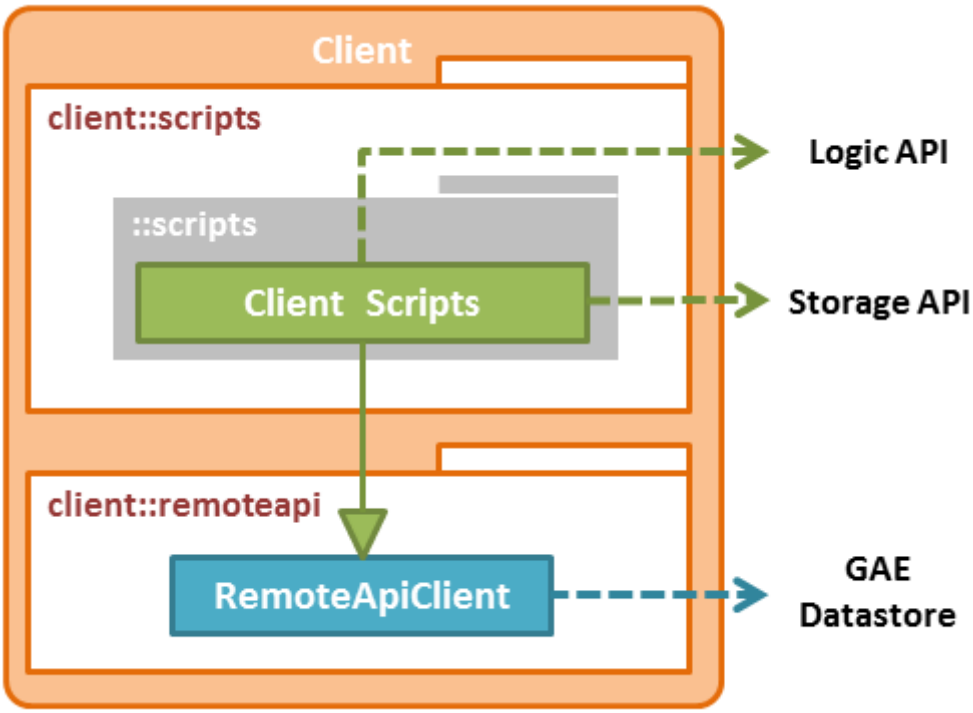


Package overview:

- `e2e.util` : Contains helpers needed for running E2E tests.
- `e2e.pageobjects` : Contains abstractions of the pages as they appear on a Browser (i.e. SUTs).
- `e2e.cases` : Contains test cases.
  - `.util` : Component test cases for testing the test helpers.
  - `.e2e` : System test cases for testing the application as a whole.
  - `.lnp` : Load and performance tests (experimental).

Client Component

The Client component contains scripts that can connect directly to the application back-end for administrative purposes, such as migrating data to a new schema and calculating statistics.



Package overview:

- `client.util` : Contains helpers needed for client scripts.
- `client.remoteapi` : Classes needed to connect to the back end directly.
- `client.scripts` : Scripts that deal with the back end data for administrative purposes.