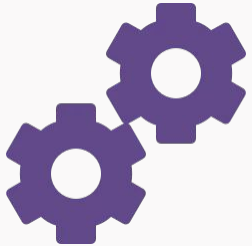
BRIGHTSPOT®

Brightspot Backend Training

Agenda



1. Demo of CMS
2. Brightspot and Dari
3. Review root styleguide concepts
4. Recipe exercise
    a. Root styleguide
    b. Data modeling
    c. Viewmodels
    d. Save lifecycle (advanced data modeling)
    e. Modifications (advanced data modeling)
    f. GraphQL API

CMS demo

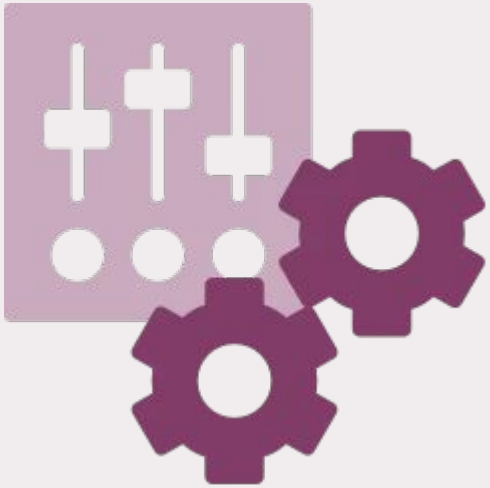
With a focus on developers

Areas of focus

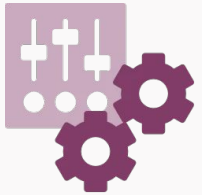
- Dashboard
- Search
- Content edit page
- Sites & Settings

Brightspot and Dari

What’s the difference?



Brightspot versus Dari



Brightspot is the CMS

- Editorial UI/UX
- Request handling and view system
- Permissions, workflows, notifications, etc.

Dari is the framework that powers Brightspot

- Database abstractions
- Data modeling using standard Java types and annotations
- Many APIs: query, file storage, tasks, etc.

Root styleguide

The glue between backend and frontend

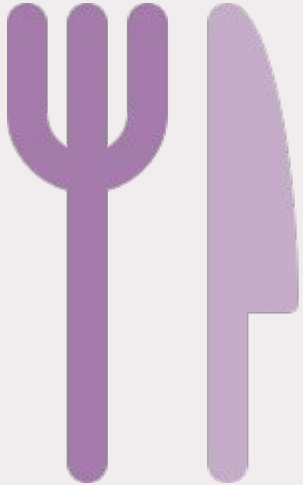
Root styleguide concepts

- Originally lived in the project root, hence the name
  - Now located in `frontend/styleguide/`, although older projects still have it in the root (`styleguide/`)
- Not to be confused with themes or bundles, which have their own `styleguide/` directories
- Specifies views, which are collections of fields, each with a specific type
- Each view requires a `.json` file and a `.hbs` file
  - The `.hbs` file should always be empty
- Groups (traditionally located in `_groups/`) allow fields to support multiple types
  - e.g. `media` field can support either an image OR a video
- Each view results in a Java interface for the backend to implement

Things to keep in mind

- Root styleguide is *only* for use with Handlebars bundles
  - If you're working with GraphQL or some other API format, you should not use the root styleguide
  - Depending on circumstances it may make sense to use the same views for both Handlebars and GraphQL/APIs

Recipe exercise



High-level requirements

- As an editor, I can...
  - Create recipes
  - Search for recipes
  - Feature recipes in an article
  - Tag recipes

Translated to Brightspot:

- Standalone recipe content type
- Recipe Article content type
  - References an existing recipe
- Recipe Module content type
  - References an existing recipe
- Recipe Tag content type
  - Referenced from recipes

Code

The training repo with all the code samples and the Docker container is available at https://github.com/perfectsense/training

Recipe exercise

Part one: Root styleguide

Recipe views

- RecipeModule
  - This is what we'd actually think of as a recipe
  - Also needs to be added to the Modules group so it can be used as a generic module anywhere those are supported
- RecipeArticlePage
  - Based on the existing ArticlePage, but includes a recipe

Recipe exercise

Part two: Frontend bundle

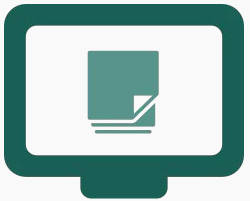
Frontend bundle

- Not covered in this session
  - See the separate frontend training for more on bundle development
- We'll apply a commit with the bundle changes so we can see our work
  - But depending on how your team functions, you might have to develop without the frontend in place

Recipe exercise

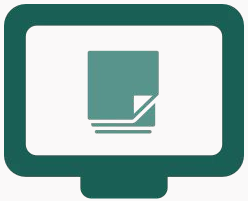
Part three: Data modeling

Data modeling basics



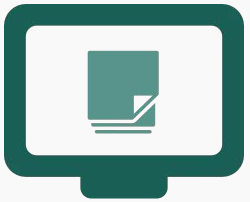
- Every content type saved to the database needs to extend `Record`
  - This enables Dari to serialize the data and save it to the database
- Depending on the use case you might extend `Content` instead
  - `Content` extends `Record` and adds full-text search, additional widgets, and UI changes
  - Anything which an editor should be able to find in CMS search should extend `Content`
  - Anything which an editor should be able to independently create should extend `Content`
- If you only want full-text search add `@Content.Searchable` and possibly `@ToolUi.ExcludeFromGlobalSearch` to your `Record` class

Data modeling: things to keep in mind



- Editorial experience is paramount
  - Make common actions as easy as possible
  - Organize fields into clusters and tabs as appropriate
  - Add placeholders and notes to guide editors
  - Hide extraneous fields and widgets
  - Ensure label is useful
    - If you have an internal name field, it should be the label
- What should be embedded?
- What should be Searchable (stored in Solr and therefore available for full-text search)
- Any need for site or global settings? Singletons?

Data modeling with annotations



Documentation:

- https://www.brightspot.com/documentation/brightspot-cms-developer-guide/latest/content-modeling-annotations
- https://www.brightspot.com/documentation/brightspot-cms-developer-guide/latest/data-modeling-annotations

Recipe exercise

Part four: Viewmodels

What is MVVM?

MVVM facilitates a **separation of development of the graphical user interface from development of the business logic or back-end logic** (the data model).

The viewmodel of MVVM is a **value converter**, meaning the viewmodel is responsible for exposing (converting) the data objects from the model in such a way that objects are easily managed and presented. In this respect, the viewmodel is **more model than view**, and **handles most if not all of the view's display logic**.

You can think of a viewmodel as a function from a model to a renderable view.

Viewmodels in Brightspot

Model: any class, but often a Recordable

Viewmodel: class which extends `ViewModel<M>`, where `M` is the model. Within a ViewModel, the model is never null (if it is, the ViewModel will not be created in the first place)

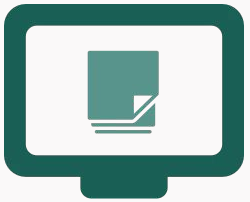
View: to the backend, simply an interface. It is bound to a specific method of rendering the actual view, e.g. HTML, JSON, etc.

Often with GraphQL we don't bother with view interfaces, but more on that later

Entry points to the view system

- `PageEntryView`
  - When serving a request, Brightspot looks for a `ViewModel` which implements `PageEntryView` and whose model is satisfied by the main content for that request
- `PreviewPageView`
  - Primarily for supporting preview of non-page content types—mainly modules
  - Used only with CMS preview

ViewModel annotations

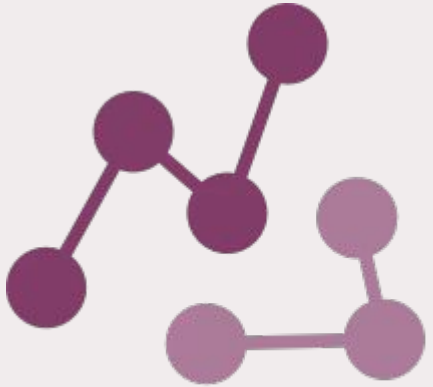


Documentation:

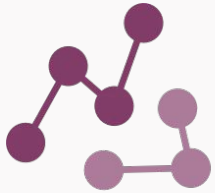
- https://www.brightspot.com/documentation/brightspot-cms-developer-guide/latest/annotations#field-annotations

Recipe exercise

Part five: Indexes and save lifecycle

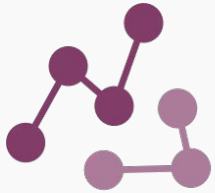


More requirements



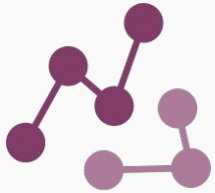
- As a user (or an editor), I can...
  - Search for recipes by difficulty; recipe tags; prep, inactive prep, cook, or total time
  - Search for recipes by typing a tag name (not just selecting a Recipe Tag in the filters)
  - Sort recipes by difficulty

Indexed methods



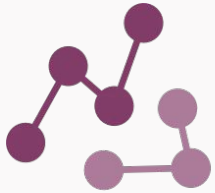
- `@Indexed` is not restricted to just fields
- Can also apply it to methods
- Indexed methods cannot accept any parameters and must be named starting with `get`, `is` or `has`
- When the record is saved, all indexed methods are called and the return value is indexed as if it had been a regular field
- Indexed methods display as a read-only field by default; typically you want to hide them with `@ToolUi.Hidden`

Save lifecycle



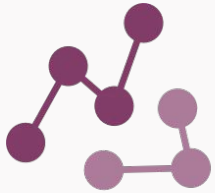
1. `beforeSave`: “on keypress” would be a more accurate name
2. `onValidate`: allows custom validation, e.g. something you can’t do with an existing annotation like `@ToolUi.Minimum`
   - Report errors through `State#addError()` (don’t throw an exception)
3. `afterValidate`: called regardless of validation success/failure
4. `beforeCommit`: only called if validation was successful
5. `onDuplicate`: allows handling uniqueness constraint violations
6. `afterSave`: called only if save was successful

Delete lifecycle



1. `beforeDelete`: self-explanatory
2. `afterDelete`: called only if the delete was successful

Save and delete lifecycles



More details at
https://www.brightspot.com/documentation/brightspot-cms-developer-guide/latest/persistence-apis

Recipe exercise

Part six: Modifications

Why Modifications?

- Java does not support multiple inheritance
  - Interfaces do not support fields
- Would like to share fields and methods among multiple content types without them having a common base class
- Modifications enable sharing fields and methods among all classes implementing an interface or extending a class

Modifications

- Abstract class which extends Record
- But not a regular content type as you might otherwise expect
- Parameterized: `abstract class Modification<T>`
- The `T` is what will be modified
  - It can be an abstract or concrete class, or an interface
- Everything extending/implementing `T` will receive all fields and methods declared in the modification class
- Still a separate class from what is being modified
  - Use `Recordable#as()` to get to the modification

Example: HasRecipes

- Common pattern to have an index shared across multiple types
  - Some types want to supply the values to index from a field, others from a method
- If you want to query across multiple types they need to share a common index
- This pattern satisfies these needs and is used frequently
  - `HasRecipes`: interface with method supplying the needed value
  - `HasRecipesData`: `Modification<HasRecipes>`with an indexed method
  - `HasRecipesWithField`: extends HasRecipes and supplies the value from an editorial field
  - `HasRecipesWithFieldData`: `Modification<HasRecipesWithField>`with the actual field

Recipe exercise

Part seven: GraphQL API

APIs versus Handlebars

- Root styleguide and views/viewmodels based on it are designed for use with Handlebars
  - They are not meant for public consumption as required by an external API
- An API requires a robust, curated, and maintainable specification
- Data format for external use is often different from what is needed in Handlebars
- Recommendation is to write custom viewmodels specifically for GraphQL (or any other external API)
  - Although depending on your use-case the Handlebars views might be appropriate

Viewmodels for GraphQL

- Almost exactly the same as viewmodels for Handlebars
- The main difference is not using auto-generated view interfaces from the root styleguide
- Typically forgo view interfaces entirely
  - You'll need interfaces if your API requires GraphQL interfaces or unions
- All public no-parameter methods of the `ViewModel` will be exposed as fields in the GraphQL API
  - Need `@ViewInterface` annotation on the viewmodel class to tell the view system to expose the methods
- For a JSON API, need `@JsonView` annotation on the viewmodel class
  - Not required for GraphQL

GraphQL entry points

- GraphQL in Brightspot does not use `PageEntryView`
- Instead, need to subclass `ContentDeliveryApiEndpoint` and explicitly specify the views/viewmodels that act as entry points to the graph
  - `PageEntryView` could be used as an entry point

GraphQL considerations

- GraphQL is an API
  - Authorization
  - CORS
  - Backwards compatibility
- Do you need to allow searching for content or is id/path sufficient?

Conclusion

Resources

- Training repo
  - https://github.com/perfectsense/training
  - Recipe exercise is on the branch `exercise/recipe`
  - Slides and recording of this presentation will be made available soon
- Brightspot documentation
  - https://www.brightspot.com/documentation
- Docker readme
  - https://hub.docker.com/r/brightspot/tomcat
- Support portal
  - Contact your product manager for access

Final questions

Ask me anything
(about Brightspot)

BRIGHTSPOT®

Best of luck developing with Brightspot!