XML Applications for Historical and Literary Research

Jonathan Martin
Scott Paul McGinnis
Welcome to DHSI 2018!

Thanks for joining the DHSI community!

In this booklet, you will find essential course materials prefaced by some useful information about getting settled initially at UVic, finding your way around, getting logged in to our network (after you’ve registered the day before our courses begin), and so on.

Given our community’s focus on things computational, it will be a surprise to no one that we might expect additional information online for some of the classes - your instructors will let you know - or that the most current version of all DHSI-related information may be found on our website at dhsi.org.

To access the DHSI wifi network, simply go into your wireless settings and connect to the “DHSI” network and enter the password “dhsi2018”.

And please don’t hesitate to be in touch with us at institut@uvic.ca or via Twitter at @AlyssaA_DHSI or @DHInstitute if we can be of any help ....
The 2018 schedule is just about ready! A very few things to confirm, add, etc, but this is the place to be to find out what is happening when / where ...

Psst: Some Suggested Outings

If you're here a day or two before we begin, or staying a day or two afterwards, here are a few ideas of things you might consider doing ....

▼ Suggested Outing 1, Botanical Beach (self-organised; car needed)
A self-guided visit to the wet, wild west coast tidal shelf (and historically-significant former research site) at Botanical Beach: we recommend departing early (around 8.00 am) to catch low tide for a better view of the wonderful underwater life! Consider bringing a packed lunch to nibble-on while looking at the crashing waves when there, and then have an afternoon drink enjoying the view from the deck of the Port Renfrew Hotel.

▼ Suggested Outing 2, Butchart Gardens (self-organised)
A shorter journey to the resplendently beautiful Butchart Gardens and, if you like, followed by (ahem) a few minutes at the nearby Church and State Winery, in the Saanich Peninsula. About an hour there by public bus from UVic, or 30 minutes by car.

▼ Suggested Outing 3, Saltspring Island (self-organised; a full day, car/bus + ferry combo)
Why not take a day to explore and celebrate the funky, laid-back, Canadian gulf island lifestyle on Saltspring Island. Ferry departs regularly from the Schwartz Bay ferry terminal, which is about one hour by bus / 30 minutes by car from UVic. You may decide to stay on forever ....

▼ Suggested Outing 4, Paddling Victoria's Inner Harbour (self-organised)
A shorter time, seeing Victoria’s beautiful city centre from the waterways that initially inspired its foundation. A great choice if the day is sunny and warm. Canoes, kayaks, and paddle boards are readily rented from Ocean River Adventures and conveniently launched from right behind the store. Very chill.

▼ And more!
Self-organised High Tea at the Empress Hotel, scooter rentals, visit to the Royal BC Museum, darts at Christies Carriage House, a hangry breakfast at a local diner, whale watching, kayaking, brew pub sampling (at Spinnaker's, Swans, Moon Under Water, and beyond!), paddle-boarding, a tour of used bookstores, and more have also been suggested!

9:00 to 4:00
▼ Early Class Meeting: 4. [Foundations] DH For Department Chairs and Deans (Hickman 120, Classroom)
Further details are available from instructors in mid May to those registered in the class. Registration materials will be available in the classroom.

3:00 to 5:00
DHSI Registration (MacLaurin Building, Room A100)
After registration, many will wander to Cadboro Bay and the pub at Smuggler's Cove OR the other direction to Shelbourne Plaza and Maude Hunter's Pub OR even into the city for a nice meal.

Monday, 4 June 2018

Your hosts for the week are Alyssa Arbuckle, Ray Siemens, and Dan Sondheim.

7:45 to 8:15
Last-minute Registration (MacLaurin Building, Room A100)

8:30 to 10:00
Welcome, Orientation, and Instructor Overview (MacLaurin A144)
Classes in Session (click for details and locations)

3. [Foundations] Making Choices About Your Data (MacLaurin D109, Classroom)
4. [Foundations] DH For Department Chairs and Deans (Hickman 120, Classroom)
5. [Foundations] Introduction to Javascript and Data Visualization (Clearihue D132, Classroom)
6. [Foundations] Introduction to Computation for Literary Criticism (Cornett A105, Lab)
7. Out-of-the-Box Text Analysis for the Digital Humanities (Human and Social Development A160, Lab)
8. Sounds and Digital Humanities (MacLaurin D111, Classroom)
9. Digital Humanities Pedagogy: Integration in the Curriculum (MacLaurin D016, Classroom)
10. Text Processing - Techniques & Traditions (McPherson Library A003, Classroom)
11. 3D Modelling for the Digital Humanities and Social Sciences (MacLaurin D010, Classroom)
12. Conceptualising and Creating a Digital Edition (MacLaurin D103, Classroom)
13. Visualizing Information: Where Data Meets Design (MacLaurin D107, Classroom)
14. Introduction to Electronic Literature in DH: Research and Practice (MacLaurin D115, Classroom)
15. Race, Social Justice, and DH: Applied Theories and Methods (MacLaurin D105, Classroom)
16. XML Applications for Historical and Literary Research (Clearihue A103, Lab)
17. 3D Modelling for the Digital Humanities and Social Sciences (Clearihue D130, Classroom)
18. Processing Humanities Multimedia (Human and Social Development A150, Lab)
19. Digital Games as Tools for Scholarly Research, Communication and Pedagogy (MacLaurin D110, Classroom)
20. Web APIs with Python (Human and Social Development A170, Lab)
21. Ethical Data Visualization: Taming Treacherous Data (MacLaurin D101, Classroom)
22. Digital Publishing in the Humanities (Clearihue D131, Classroom)
23. Linked Open Data and the Semantic Web (Clearihue D130, Classroom)
24. Introduction to IIIF: Sharing, Consuming, and Annotating the World’s Images (MacLaurin D114, Classroom)
25. Feminist Digital Humanities: Theoretical, Social, and Material Engagements (Cornett A229, Classroom)
26. The Frontend: Modern JavaScript & CSS Development (Clearihue A030, Classroom)
27. Historical and Literary Practice (Clearihue A109, Classroom)
28. Digital Humanities Pedagogy: Integration in the Curriculum (MacLaurin D016, Classroom)
29. Text Processing - Techniques & Traditions (McPherson Library A003, Classroom)
30. 3D Modelling for the Digital Humanities and Social Sciences (MacLaurin D010, Classroom)
31. Conceptualising and Creating a Digital Edition (MacLaurin D103, Classroom)
32. Visualizing Information: Where Data Meets Design (MacLaurin D107, Classroom)
33. Introduction to Electronic Literature in DH: Research and Practice (MacLaurin D115, Classroom)
34. Race, Social Justice, and DH: Applied Theories and Methods (MacLaurin D105, Classroom)
35. XML Applications for Historical and Literary Research (Clearihue A103, Lab)
36. Processing Humanities Multimedia (Human and Social Development A150, Lab)
37. Digital Games as Tools for Scholarly Research, Communication and Pedagogy (MacLaurin D110, Classroom)
38. Web APIs with Python (Human and Social Development A170, Lab)
39. Ethical Data Visualization: Taming Treacherous Data (MacLaurin D101, Classroom)
40. Digital Publishing in the Humanities (Clearihue D131, Classroom)
41. Linked Open Data and the Semantic Web (Clearihue D130, Classroom)
42. Introduction to IIIF: Sharing, Consuming, and Annotating the World’s Images (MacLaurin D114, Classroom)
43. Feminist Digital Humanities: Theoretical, Social, and Material Engagements (Cornett A229, Classroom)
44. The Frontend: Modern JavaScript & CSS Development (Clearihue A030, Classroom)

10:15 to Noon

Lunch break / Unconference Coordination Session (MacLaurin A144)
(Grab a sandwich and come on down!)

Undergraduate Meet-up, Brown-Bag (details via email)

1:30 to 4:00

Classes in Session

Institute Panel: Perspectives on DH (or, #myDHis …)
Chair: Alyssa Arbuckle (U Victoria)
(MacLaurin A144)

▼ Milena Radzikowska (Mt Royal C): "Release the Kraken: Story-Driven Prototyping for the Digital Humanities."
Abstract: I have spent the last 15 years of my career designing text analysis tools for use by humanities scholars. In this brief presentation, I propose to share a concept-based approach to interface design for DH.

▼ Emily Murphy (U Victoria): "#MyDHis Edgy."
Abstract: I will build upon—or, possibly, perform a misprision of—a tweet by Polina Vinogradova; "#myDHis messy, dusty, edgy, and radically inclusive!" Vinogradova evokes the mess and dust of the archives, the edges that connect nodes of a network, and the political impetus to think of cultural history and community together. I argue that these aspects of DH have a renewed importance as we head into a moment of feminist historiography.

▼ Margaret Konkol (Old Dominion U): "Prototyping Mina Loy’s Alphabet with a 3D Printer."
Abstract: This talk discusses the interpretive and methodological implications of using 3D printing technologies to prototype the archival diagrams of a proposed but never constructed plastic segmental alphabet letter kit—a game designed by modernist poet Mina Loy for F.A.O Schwarz. Although intended as a toy for young children, "The Alphabet that Builds Itself," as a work of "object typography" articulates a theory of language as kinetic, geometric, recombinant, and open to mutation. Alphabetic segments extend into the x, y, and z coordinates in exponential iterations and conjoin with magnets. Combining elements of contemporaneous typefaces like Futura and Gill Sans, which represented modernity’s functional ideals and democratic principles of simplicity, these recombinant letters represent, as this talk argues, Loy’s unpublished modernist poem, an articulation of Loy’s concept of language as a physical fact in which substance, not just form, is semantic.

▼ Lee Zickel (Case Western Reserve U): "Comfortably Trepid."
Abstract: #myDHs found outside the well-established, DH-friendly institutions, at an institution that is devoted predominantly to Medicine and Engineering. I, and with increasing frequency other DH practitioners and instructors, am not positioned in a DH Lab or Humanities Center, but in ITS. Part teacher, part technologist, part translator, I will briefly discuss my work supporting humanists and social scientists, particularly those who are new to or less comfortable with computational methodologies.

▼ Dorothy Kim (Vassar C): "#MyDHis Antifascist."
Abstract: I’ve spent a lot of time in the last 12 months thinking about fascism, digital humanities, its long histories, and what it means to do DH work that centers social justice particularly in this global rise of late fascism. I will speak briefly about DH’s history, including the medieval history related to Busa but how that history really connects to data systems that created the Holocaust and also participated in the Cold War nuclear military complex.
Randa El Khatib (U Victoria): "Learning from the Iterative Process."
Abstract: #MyDHis Iterative. In addition to the improvements that come with iterative projects, the iterative process itself is a fruitful area for scholarly inquiry. Within this iterative context, various teams that I work with and have been reflecting on and rethinking central DH practices, such as what it means to collaborate, prototype, remix, and implement DH values in our work. In this talk, I will present the various lessons learnt along the way.

Sarah Melton (Boston C): "#MyDHis...People."
Abstract: Taking seriously Miriam Posner’s exhortation to “commit to DH people, not DH projects,” I invite us to reflect on how people are the core of DH. In this brief talk, I will explore the intersections between DH, labor, and infrastructure.

Opening Reception (University Club)
We are grateful to Gale Cengage for its sponsorship.

Tuesday, 5 June 2018

9:00 to Noon
Classes in Session

12:15 to 1:15
Lunch break / Unconference
"Mystery" Lunches
DHSI Lunchtime Workshop Session (click for workshop details and free registration for DHSI participants)
- 73. Introduction to ORCID (Digital Scholarship Commons, Classroom).

1:30 to 4:00
Classes in Session

DHSI Colloquium Lightning Talk Session 1 (MacLaurin A144)
Chair: James O’Sullivan
- New Modes of DH and Archival Skills Acquisition in a Graduate Public History Course. Paulina Rousseau (Ryerson U)
- Walking a Transect: Exploring a Soundscape. John Barber (Washington State U)
- Centering the Edge Case: Designing Services for Humanities Data Research. Grace Afsari-Mamagani (New York U)
- Orwellian Vocabulary and the 21st-Century Politics. Ilgin Kizilgunesler (U Manitoba)
- Making Open Data from a Gray Archive. Sara Palmer (Emory U)

6:00 to 8:00
DHSI Newcomer’s Beer-B-Q (Felicitas, Student Union Building)

Wednesday, 6 June 2018

9:00 to Noon
Classes in Session

12:15 to 1:15
Lunch break / Unconference
"Mystery" Lunches
Brown Bag Lecture: Alexandra Branzan Albu (U Victoria): “Visual Recognition of Symbolic and Natural Patterns” (Digital Scholarship Commons, 3rd Floor McPherson Library)
Abstract: Image-based object recognition is a visual pattern recognition problem; one may characterize visual patterns as either symbolic or natural. Symbolic patterns evolved for human communication; they include but are not limited to text, forms, tables, graphics, engineering drawings etc. Symbolic patterns vary widely in terms of size, style, language, alphabet and fonts; however, literate humans can easily compensate for this variability and instantly recognize most symbolic patterns. On the other hand, natural patterns characterize images of physical structures; they often lack the intrinsic discriminability and structure of symbolic patterns, and vary widely in terms of pose, perspective, and lighting.

This lecture will explore similarities and differences in approaches designed for recognizing visual and symbolic patterns, and will address the following questions via examples.
- What are the distinctive characteristics of natural patterns? What dimensions of variability can we infer?
- What are the distinctive characteristics of symbolic patterns? What dimensions of variability can we infer?

Alexandra Branzan Albu is an Associate Professor with the Department of Electrical and Computer Engineering and cross-listed with Computer Science. Her research interests are related to image analysis, computer vision, and visual computing. She is actively pursuing outreach activities dedicated to increasing the women's presence in electrical engineering and computer science.
Thursday, 7 June 2018

9:00 to Noon  Classes in Session

12:15 to 1:15  Lunch break / Unconference
   "Mystery" Lunches

1:30 to 4:00  Classes in Session

4:15 to 5:15  DHSI Colloquium Lightning Talk Session 2  (MacLaurin A144)
   Chair: James O’Sullivan
   ● Defining a Taxonomy of Abandonment for Online Digital Humanities Projects. Luis Meneses (Electronic Textual Cultures Lab, U Victoria) and Jonathan Martin (King’s College London)
   ● The Stories We Tell: Representing Gay and Lesbian History through Digital Technologies in the LGCLC Project. Nadine Boulay (Simon Fraser University) and Ewan Matthews (Ryerson U)
   ● Italian Paleography in the Digital Domain. Isabella Magni (Newberry Library)
   ● Digital Humanities, A Question of Ethics. Negar Basiri (Louisiana State U)
   ● Writing Poetry in High School. Guadalupe Echegoyen (National Autonomous U Mexico)

6:00 to 7:00  “Half Way There!” [An Informal, Self-Organized Birds of a Feather Get-Together]  (Felicitas, Student Union Building)
   Bring your DHSI nametag and enjoy your first tipple on us!

7:30 to 9:30  (Groovy?) Movie Night  (MacLaurin A144)

Friday, 8 June 2018 [DHSI; DLFxDHSI Opening]

9:00 to Noon  DHSI Classes in Session

12:15 to 1:15  DHSI Lunch Reception / Course E-Exhibits  (MacLaurin A100)

1:00 to 2:00  DLFxDHSI Registration  (MacLaurin A100)

1:30 to 1:50  [DHSI] Remarks, A Week in Review  (MacLaurin A144)

2:00 to 3:00  Joint Institute Lecture (DHSI and DLFxDHSI):
   Bethany Nowviskie (CLIR DLF and U Virginia): “Reconstitute the World: Machine-reading Archives of Mass Extinction”
   Chair: Lisa Goddard (U Victoria)  (MacLaurin A144)
   Abstract: The basic constitution of our digital collections becomes vastly more important in the face of two understandings: first, that archives of modernity are archives of the sixth great mass extinction of life on our planet; and next, that we no longer steward cultural heritage for human readers alone. In the same way that we people are shaped by what we read, hear, and see, the machine readers that follow us into and perhaps beyond the Anthropocene have begun to learn from "unsupervised" encounters with our digital libraries. What will we preserve for the living generations and artificial intelligences that will come? What do we neglect, or even choose to extinguish? And from an elegiac archive, a library of endings, can we create forward-looking, speculative collections—collections from which to deep-dream new futures? The most extra/ordinary power we possess is the power to make poetry from records of the past. Could it be called on, one day, to reconstitute the world?
### 3:30 to 5:00
- Joint Reception: DHSI and DLFxDHSI *(University Club)*
- DLFxDHSI Poster/Demo Session
  - DHSI Colloquium Poster/Demo Session
    - Mediers as a Colonialist Artifact in Menzies’ Journal. Paula Johanson (U Victoria)
    - Camp Edit: the Institute for the Editing of Historical Documents. Nikolaus Wasmoen (Association for Documentary Editing, U Buffalo), Jennifer Stiertzer (Association for Documentary Editing, U Virginia), and Cathy Moran Hajo (Association for Documentary Editing, Ramapo C)
    - A Digital Archaeology of Life in Cleveland’s Depression-Era Slums. Charlie Harper (Case Western Reserve U) and Jared Bendis (Case Western Reserve U)
    - Feminist Pest Control: controlling and not controlling nonhuman pests. Lindsay Garcia (C of William and Mary)
    - Legends of the Buddhist Saints. Jonathan S. Walters (Whitman C) and Dana Johnson (Freelance Web Developer)
    - Accessibility in Digital Environments Via TEI-Encoded Uncontracted Braille. Gia Alexander (Texas A&M U)
    - Translation3point0: Why Literary Translation Data Matters. Katie King (U Washington)
    - PoéticaSonora: A Digital Audio Repository Prototype for Latin American Sound Art and Poetry. Aurelio Meza (Concordia U)
    - Beauty and the Book: Pre-Raphaelite Artistic Practice Contained. Josie Greenhill (U Victoria)
    - Poetic Procedures/Digital Deformances. Corey Sparks (California State U, Chico)
    - Miranda, the Folger Shakespeare Library’s new Digital Asset Platform. Meaghan Brown (Folger Shakespeare Library)
    - Living Song Project. Quinn Patrick Ankrum (U Cincinnati) and Elizabeth Avery (U Oklahoma)
    - Digital Frankenstein Variorum. Rikk Mulligan (Carnegie Mellon U)

### Saturday, 9 June 2018 [DLFxDHSI + DHSI Conference and Colloquium]

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>8:30 to 9:00</td>
<td>DLFxDHSI Registration <em>(MacLaurin A100)</em></td>
</tr>
<tr>
<td>9:00 to 5:30</td>
<td>DLFxDHSI UnConference Sessions</td>
</tr>
<tr>
<td>9:00 to 4:00</td>
<td><em>DHrS All Day Workshop Session</em> <em>(click for workshop details and free registration for DHSI participants)</em></td>
</tr>
<tr>
<td>9:00 to 5:00</td>
<td><em>DHSI Colloquium Day Conference</em> <em>(MacLaurin A144)</em></td>
</tr>
</tbody>
</table>

---

**Welcome**

*People I: Documenting Online Lives. Chair: Molly Nebiolo (University of New York)*

- Examining Gendered Harassment Online and in Silicon Valley. Andrea Flores (Utica College)
- This is Just to Say I Have `<X>` the `<Y>` in your `<Z>`: Modernist Memes in an Era of Public Apology. Shawna Ross (Texas A&M University)

**Break**

*People II: Documenting Lives Online. Chair: Dheepa Sundaram (College of Wooster)*

- Youtube Yoga and Ritual on Demand: The Virtual Economics of Hindu Soteriology. Dheepa Sundaram (College of Wooster)
- The Resemblage Project: Creativity and Digital Health Humanities in Canada. Andrea Charise (University of Toronto) and Stefan Krecsy (University of Toronto)

**Lunch**

*Projects I: Building and Analyzing. Chair: Yannis Rammos (New York University)*

- Building the ARTECHNE Database: New directions in Digital Art History. Marieke Hendriksen (Old Dominion University)
- The Ineffective Inquisition: The Holy Office’s Sphere of Influence in Early Modern New Spain. Kira Homo (Pennsylvania State University)

**Break**

*Projects II: Mapping and Visualizing. Chair: Innocent Opara (Qumet Institute)*

- Mapping Sarah Sophia Bank’s Numismatic Collection. Erica Hayes (North Carolina State University) and Kacie Wills (University of California, Riverside)
- Text Mining and Visualizing 18th Century American Correspondence. Ashley Sanders Garcia (University of California, Los Angeles)

**Break**

*Practices: Digital Scholarship on Campus and in the Classroom. Chair: Alyssa Arhuckle (University of Victoria)*
### Sunday, 10 June 2018 [SINM + DHSI Registration, Workshops]

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>8:30 to 9:00</td>
<td>Symposium on Indigenous New Media Registration (<a href="#">MacLaurin A100</a>)</td>
</tr>
<tr>
<td>9:00 to 5:00</td>
<td>DHSI Registration (<a href="#">MacLaurin A100</a>)</td>
</tr>
<tr>
<td>9:00 to 4:00</td>
<td><strong>SINM Sessions</strong></td>
</tr>
<tr>
<td></td>
<td>• 63. Symposium on Indigenous New Media: Reading Group (<a href="#">Hickman 105, Classroom</a>)</td>
</tr>
<tr>
<td></td>
<td>• 72. Symposium on Indigenous New Media: Indigitization (<a href="#">Hickman 120, Classroom</a>)</td>
</tr>
<tr>
<td></td>
<td><a href="#">Full details here</a></td>
</tr>
<tr>
<td>9:00 to 4:00</td>
<td><strong>DHSI All Day Workshop Sessions</strong> (<a href="#">click for workshop details and free registration for DHSI participants</a>)</td>
</tr>
<tr>
<td></td>
<td>• 53. Building Your Academic Digital Identity (<a href="#">MacLaurin D105, Classroom</a>)</td>
</tr>
<tr>
<td></td>
<td>• 54. An Introduction to the Archaeology of 1980s Computing (<a href="#">MacLaurin D114, Classroom</a>)</td>
</tr>
<tr>
<td>9:00 to Noon</td>
<td><strong>DHSI AM Workshop Sessions</strong> (<a href="#">click for workshop details and free registration for DHSI participants</a>)</td>
</tr>
<tr>
<td></td>
<td>• 55. Regular Expressions (<a href="#">MacLaurin D111, Classroom</a>)</td>
</tr>
<tr>
<td></td>
<td>• 56. 3D Visualization for the Humanities (<a href="#">MacLaurin D010, Classroom</a>)</td>
</tr>
<tr>
<td></td>
<td>• 58. DH Fieldwork Methods (<a href="#">MacLaurin D016, Classroom</a>)</td>
</tr>
<tr>
<td></td>
<td>• 60. Pedagogy of the Digitally Oppressed: Inculcating De-/Anti-/Post-Colonial Digital Humanities (<a href="#">MacLaurin D107, Classroom</a>)</td>
</tr>
<tr>
<td></td>
<td>• 61. Introduction to #GraphPoem. Digital Tools for Poetry Computational Analysis and Graph Theory Apps in Poetry (<a href="#">MacLaurin D101, Classroom</a>)</td>
</tr>
<tr>
<td></td>
<td>• 62. Creating a CV for Digital Humanities Makers (<a href="#">MacLaurin D115, Classroom</a>)</td>
</tr>
<tr>
<td>1:00 to 4:00</td>
<td><strong>DHSI PM Workshop Sessions</strong> (<a href="#">click for workshop details and free registration for DHSI participants</a>)</td>
</tr>
<tr>
<td></td>
<td>• 64. Agent-Based Modelling in the Humanities (<a href="#">MacLaurin D111, Classroom</a>)</td>
</tr>
<tr>
<td></td>
<td>• 65. Unleash Linux on MacOS (<a href="#">MacLaurin D010, Classroom</a>)</td>
</tr>
<tr>
<td></td>
<td>• 66. DHSI Knits: History of Textiles and Technology (<a href="#">MacLaurin D016, Classroom</a>)</td>
</tr>
<tr>
<td></td>
<td>• 67. Crowdsourcing as a Tool for Research and Public Engagement (<a href="#">MacLaurin D109, Classroom</a>)</td>
</tr>
<tr>
<td></td>
<td>• 69. Web Annotation as Critical Humanities Practice (<a href="#">MacLaurin D103, Classroom</a>)</td>
</tr>
<tr>
<td></td>
<td>• 70. Dynamic Ontologies for the Humanities (<a href="#">MacLaurin D107, Classroom</a>)</td>
</tr>
<tr>
<td></td>
<td>• 71. Social Media Research in the Humanities (<a href="#">MacLaurin D101, Classroom</a>)</td>
</tr>
<tr>
<td>4:10 to 5:00</td>
<td><strong>Joint Institute Lecture (DHSI and SINM):</strong></td>
</tr>
</tbody>
</table>
|          | David Gaertner (U British Columbia): *"A Landless Territory?: CyberPowWow and the Politics of Indigenous New Media."
|          | Chair: Deanna Reder (Simon Fraser U) ([MacLaurin A144](#))                                |
|          | [Abstract](#): Following the 1997 launch of Skawennati’s (Mohawk) CyberPowWow, digital space has become a vital new territory for the resurgence of Indigenous storytelling and cultural practice: "We have signed a new treaty," Cree artist Archer Pechawis wrote of this period, "and it is good. We have the right to hunt, fish, dance and make art at www.CyberPowWow.net, .org and .com for as long as the grass grows and the rivers flow." This talk will critically explore the theoretical, cultural, political-economic, and gendered dynamics underlying the histories and futures of Indigenous new media. Particular attention will be given in examining the ways in which new media and digital storytelling connect to and support key issues in the field of Indigenous studies, such as sovereignty, self-determination, decolonization, and land rights. |

### Monday, 11 June 2018 [DHSI + SINM]

After the day, many will wander to [Cadboro Bay](#) and the pub at [Smuggler's Cove](#) OR the other direction to [Shelbourne Plaza](#) and [Maudie Hunter's Pub](#) OR even into the city for a [bite to eat](#).
Your hosts for the week are Ray Siemens and Dan Sondheim.

7:45 to 8:15
DHSI Last-minute Registration (MacLaurin A100)

8:30 to 10:00
DHSI Welcome, Orientation, and Instructor Overview (MacLaurin A144)

9:00 to 4:00
SINM Sessions

- DHSI Classes in Session (click for details and locations)
  - 29. [Foundations] Models for DH at Liberal Arts Colleges (& 4 yr Institutions) (MacLaurin D109, Classroom)
  - 32. Stylometry with R: Computer-Assisted Analysis of Literary Texts (Clearihue A102, Lab)
  - 33. Digital Storytelling (MacLaurin D111, Classroom)
  - 34. Text Mapping as Modelling (Clearihue D131, Classroom)
  - 35. Geographical Information Systems in the Digital Humanities (Clearihue A105, Lab)
  - 36. Open Access and Open Social Scholarship (MacLaurin D114, Classroom)
  - 37. Introduction to Machine Learning in the Digital Humanities (Cornett A229, Classroom)
  - 38. Queer Digital Humanities: Intersections, Interrogations, Iterations (MacLaurin D110, Classroom)
  - 40. Using Fedora Commons / Islandora (Human and Social Development A160, Lab)
  - 41. Documenting Born Digital Creative and Scholarly Works for Access and Preservation (MacLaurin D115, Classroom)
  - 42. Games for Digital Humanists (MacLaurin D016, Classroom & Human and Social Development A170, Lab)
  - 43. XPath for Document Archeology and Project Management (Cornett A128, Classroom)
  - 44. Surveillance and the Digital Humanities (MacLaurin D103, Classroom)
  - 73. Introduction to ORCID (Digital Scholarship Commons, Classroom)

10:15 to Noon
Joint Institute Lecture (DHSI and SINM):
Jordan Abel (Simon Fraser U): “Indigeneity, Conceptualism, and the Borders of DH.”
Chair: Michelle Brown (U Hawaii)
(MacLaurin A144)

Abstract: This talk brings together digital humanities discourses in computational textual analysis and Indigenous Literary Studies to analyze a corpus comprised of every book of Indigenous poetry published in Canada, extending from Pauline Johnson's 1895 book The White Wampum to Marilyn Dumont's 2015 book The Pemmican Eaters. While the main goal of this research project initially centered on the topic modeling of a corpus of Indigenous poetry, the project also addresses the systemic barriers that have prevented such work gaining traction, and likewise attempts to address the specific challenges that Indigenous writing (and in particular Indigenous poetry) present to current Digital Humanities methodologies.

12:15 to 1:15
Lunch break / Unconference Coordination Session (MacLaurin A144)
(Grab a sandwich and come on down!)
DHSI Undergraduate Meet-up, Brown-Bag (details via email)

1:30 to 4:00
DHSI Classes in Session

- Joint Institute Lecture (DHSI and SINM):
  Jordan Abel (Simon Fraser U): “Indigeneity, Conceptualism, and the Borders of DH.”
  Chair: Michelle Brown (U Hawaii)
  (MacLaurin A144)

4:10 to 5:00
Joint Reception: DHSI and SINM (University Club)

Tuesday, 12 June 2018

9:00 to Noon
Classes in Session

12:15 to 1:15
Lunch break / Unconference
Mystery” Lunches
DHSI Lunchtime Workshop Session (click for workshop details and free registration for DHSI participants)

- 73. Introduction to ORCID (Digital Scholarship Commons, Classroom).

9:00 to Noon
Classes in Session

12:15 to 1:15
Lunch break / Unconference
Mystery” Lunches
DHSI Lunchtime Workshop Session (click for workshop details and free registration for DHSI participants)
Wednesday, 13 June 2018

1:30 to 4:00

- DHSI Colloquium Lightning Talk Session 4 (MacLaurin A144)
  Chair: Lindsey Seatter
  - Mapping Indigenous and Chicana/o Environmental Imaginaries using GIS. Stevie Ruiz (California State U, Northridge), Quetzalli Enrique (California State U, Northridge), Enrique Ramirez (California State U, Northridge), and Tomas Figueroa (California State U, Northridge)
  - "But is it any good?": A quantitative approach to the popularity of digital fanfiction. Suzanne Black (University of Edinburgh)
  - The American Prison Writing Archive (APWA). Doran Larson (Hamilton C), Janet Simons (Digital Humanities Initiative, Hamilton C), and William Rasenberger (Hamilton C)

4:15 to 5:15

- Doing DH with Graphic Narratives. John Barber (Washington State U)

6:00 to 8:00

- DHSI Newcomer's Beer-B-Q (Felicitas, Student Union Building)

Thursday, 14 June 2018

9:00 to Noon

- Classes in Session

12:15 to 1:15

- Lunch break / Unconference
  - "Mystery" Lunches

1:30 to 4:00

- DHSI Colloquium Lightning Talk Session 5 (MacLaurin A144)
  Chair: Lindsey Seatter
  - Faraway, so close: Has the political environment really changed in Ecuador?. Luis Meneses (Electronic Textual Cultures Lab, U Victoria)
  - Re-mixing Melville's Reading: Text Analysis of Marginalia with R and XSLT. Christopher Ohge (U London, School of Advanced Study) and Steven Olsen-Smith (Boise State U)
  - Developing Interactive and Open-Source OER: Inquiry-Based Music Theory. Evan Williamson (U Idaho)
  - Spatial Humanities and the Web of Everywhere. Ken Cooper (SUNY Geneseo)

4:15 to 5:15

- "Half Way There (yet again)!" [An Informal, Self-Organized Birds of a Feather Get-Together] (Felicitas, Student Union Building)
  Bring your DHSI nametag and enjoy your first tipple on us!

Friday, 15 June 2018

9:00 to Noon

- Classes in Session

12:15 to 1:15

- Lunch Reception / Course E-Exhibits (MacLaurin A100)

7:30 to 9:30

- (Groovier?) Movie(r) Night (MacLaurin A144)
1:30 to 2:30

(MacLaurin A144)

Abstract: Much has changed and continues to change in digital humanities since the formal establishment of Iter in the Fall of 1997. However, the mandate of the not-for-profit partnership to support “the advancement of learning in the study and teaching of Middle Ages and Renaissance (400–1700) through the development and distribution of online resources” continues to have relevance. This presentation explores the striking challenges faced by Iter and presents our current thinking on the realization of this mandate for the future through a platform with a focus on facilitating the discovery of the academic resources necessary to our work; creating an environment for collaboration, sharing and developing projects; and on enabling the distribution and publication of our scholarship.

2:40 to 3:00

Awards and Bursaries Recognition
Closing, DHSI in Review (MacLaurin A144)

Contact info:
institut@uvic.ca  P: 250-472-5401  F: 250-472-5681
Course Schedule

<table>
<thead>
<tr>
<th>Course Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction to eXist-db: Siegel and Retter, ch.1 “Introduction”</td>
<td>2</td>
</tr>
<tr>
<td>Installing eXist-db: Siegel and Retter, ch.2 “Getting Started”</td>
<td>4</td>
</tr>
<tr>
<td>Basics of eXist-db: Siegel and Retter, ch.3 “Using eXist 101”</td>
<td>22</td>
</tr>
<tr>
<td>Walmsley ch.1 “Introduction to XQuery”</td>
<td>41</td>
</tr>
<tr>
<td>Walmsley ch.2 “XQuery Foundations”</td>
<td>67</td>
</tr>
<tr>
<td>Walmsley ch.3 “Expressions: XQuery Building Blocks”</td>
<td>81</td>
</tr>
<tr>
<td>Siegel and Retter, ch.6 “XQuery for eXist”</td>
<td>97</td>
</tr>
<tr>
<td>Julia Flanders “TEI Cribsheet”</td>
<td>114</td>
</tr>
<tr>
<td>Syd Bauman “XPath Tree Diagram”</td>
<td>130</td>
</tr>
<tr>
<td>Some Online References and Resources (annotated)</td>
<td>138</td>
</tr>
<tr>
<td>Some Online References and Resources (annotated)</td>
<td>139</td>
</tr>
</tbody>
</table>

There are a few things we would like you to do before the course begins. For information, see [https://www.dhlinux.org/dhsi-xml/](https://www.dhlinux.org/dhsi-xml/).
Course Schedule

See https://www.dhlinux.org/dhsi-xml/ for the most up-to-date course schedule.

Monday, June 4
Session I: 10:15 to 12:00pm
- Course Introduction / Software Setup (if not completed before class, see welcome e-mail)
- XML Fundamentals
- Basics of HTML5

Session II: 1:30pm to 4:00pm
- Basics of HTML5
- Other XML Flavors: Exercises in KML and TEI
  [optional: library metadata example, SVG]

Tuesday, June 5
Session III: 9:00am to 12:00pm
- Advanced XML
- Basics of XPath and XQuery

Session IV: 1:30pm to 4:00pm
- XQuery Exercise

Wednesday, June 6
Session V: 9:00am to 12:00pm
  [optional: Getting deeper into HTML5]
- Introduction to eXist-db
- No-SQL and XML databases: [Key concepts / a quick overview]

Session VI: 1:30pm to 4:00pm
- Overflow
- Individual Projects
  [participants introduce their projects briefly]

Thursday, June 7
Session VII: 9:00am to 12:00pm
- Individual Projects

Session VIII: 1:30pm to 4:00pm
- Individual Projects
**Friday, June 8**

*Session IX: 9:00am to 12:00pm*
- Individual Projects
- Class Project / Show-and-Tell

**Other Helpful Information:**

There are a few things we would like you to do before the course begins. For information, see [https://www.dhlinux.org/dhsi-xml/](https://www.dhlinux.org/dhsi-xml/).

For this course, we will be using Slack. Our Slack channel is [https://xml-dh.slack.com/home](https://xml-dh.slack.com/home).

Be sure to check out the resources at the end of the coursepack!
What Is eXist?

As it turns out, this is quite a difficult question to answer. The problem lies in the wide audience that eXist serves. eXist is many things to many people, and thus there is no single succinct answer.

eXist is an open source piece of software written in Java that is freely available in both source code and binary form. eXist has always been made available under the Lesser GNU Public License (LGPL), version 2.1. While eXist makes use of many other open source libraries itself, all of these are compatible with the LGPL, and eXist eschews the GPL license in favor of freedom of choice for its users.

eXist was conceived as a native XML database. As a database, its unit of atomicity is the document, so we could very easily brand it a NoSQL document database. However, to do so would be to do an injustice to the software, and worse, to all of those who have contributed to making eXist much more than just a NoSQL database over the years.

Unlike most NoSQL databases, which each have their own proprietary database query language, eXist makes use of a standardized query language developed by the W3C: XML Query Language (XQuery). With a standard query language, you have the ability to write code that can be used not just on eXist, but on any platform or processor that supports XQuery. Some of the benefits of XQuery are that it is:

Synergistic

XQuery was carefully designed and evolved over a six-year period by an open working group with many contributors, meaning that many real industry use cases were considered during its construction. XQuery has been influenced by
several previous languages and concepts, such as Perl, Lisp, Haskell, SQL, and many more.

*Easy to use*
XQuery was designed to be simple to use and debug, meaning that nonprogrammers (given an understanding of their documents) should be able to easily construct queries. Many eXist users work in the humanities and have no formal computer science background, but are quite comfortable writing complex XQueries to query their documents.

*Easy to optimize*
The XQuery specification does not detail how an implementation should perform query processing, and its developers have given great thought to ensuring that any implementation can optimize processing of query operations. Likewise, as a moderate user of XQuery, you can often easily understand why a particular query is slow and what you may do to improve it.

*Easy to index*
Join operations in XQuery (e.g., predicates and `where` clauses) lend themselves well to index optimization, which eXist exploits to speed up XQuery execution. eXist provides multiple indexing schemes that the user may configure.

*Turing complete*
XQuery is more than just a query language: it is in a class of languages known as *Turing complete*, which means that it is a complete programming language and any program can be expressed in it. XQuery is also a functional programming language, as opposed to a procedural one, meaning that it is generally easier to construct programs that you can easily understand and that ultimately contain fewer bugs. In eXist you can build entire applications in just XQuery!

*Query first*
While XQuery is a programming language, it is designed primarily as a query language. Therefore, it is much easier to extract just a few elements from large data collections with XQuery than, say, with XSLT.

*More than you realize!*
While XQuery is easy to get started with, its functional nature can make it tricky to work with if you have only procedural programming experience. XQuery can be incredibly elegant, and we are frequently surprised at how very complex problems may be solved quite simply in XQuery when considered in a different light.

We should also perhaps mention that eXist is not just for XML documents. You can, in fact, store any file into the database, and eXist can do some very clever things with content extraction and metadata with non-XML documents to help you query and manage those binary formats too.
We could stop there and focus the rest of the book on the database, but you would really miss out on the good stuff. eXist is also a web server: you can make web requests directly to the database to store, retrieve, or update XML documents. eXist achieves this by providing an HTTP REST API that describes the database. It also provides a WebDAV interface so that your users can easily drag and drop documents from their desktops into the database, or open a document for editing.

But wait, there’s more! As eXist evolved over the years it became clear that being able to store, retrieve, and edit documents via the Web was neat, but also being able to store XQuery into the database and execute it via a web request from your web browser meant you could easily construct very powerful web applications directly on top of the database. eXist of course continued to evolve here, providing new features for forms, web application packaging, improved security, SQL queries, SSL, and support for producing and consuming JSON and HTML 5, among other offerings.

So, in summary, what is eXist?

- A NoSQL document database for XML and binary (including text)
- A web server for consuming and serving documents
- A document search engine
- A web application platform
- A document creation and capture platform (XForms)
- A data mashup and integration platform
- An embeddable set of libraries for use in your own applications
- And much, much more

**eXist Compared to Other Database Systems**

Let’s take a moment to discuss some of the main differences between eXist and other SQL and NoSQL database systems. eXist is:

*Document oriented*

Unlike traditional RDBMSs (Relational Database Management Systems) such as Oracle, MySQL, and SQL Server, which are table oriented, eXist is a NoSQL document-oriented database.

Many other NoSQL document databases (including MongoDB and Apache Cassandra) store JSON documents, whereas eXist stores XML documents. One of the key advantages of XML over JSON is the ability to handle complex document structures using *mixed content*. JSON easily handles data-oriented documents, while XML easily handles both data-oriented markup and text-oriented
documents. Another key advantage of XML over JSON is that you can adopt *namespaces* to cleanly model different business domains.

**Schemaless**

RDBMSs and even several NoSQL databases require you to define your data schema before you can start storing your data. eXist is entirely flexible, and allows you to store your documents without specifying any schema whatsoever. It is ideal for business problems that have high-variability data and also helps developers rapidly prototype and evolve applications. However, *schemaless* should not be confused with providing *validation of documents*. Should you wish, you can also define a schema in eXist and have eXist enforce that only documents meeting your schema requirements are stored or updated.

**Portable queries**

RDBMSs typically use a standardized SQL query language; however, in practice, apart from the most basic queries it can be hard to run the same SQL queries across different RDBMS database products. Likewise, most NoSQL systems have their own proprietary query languages, which are entirely product-specific. eXist takes a very different approach and provides XQuery and XSLT, which are W3C standardized query and transformation languages, meaning that with very little effort you can execute your eXist queries on any other product that provides an XQuery and/or XSLT processor.

**Structured search**

Like many database systems, eXist allows you to define different indexes for your searches. However, combined with the ability to search based on the document structure, this makes eXist search results more precise than those of almost any other database when dealing with structured documents such as TEI, DocBook, and DITA. eXist will consistently have better search metrics (precision and recall) than search systems that ignore document structure. If findability is high on your list of desired attributes, then eXist is a great choice.

**Forms**

Oracle provides Oracle forms for use with its RDBMS. We are not aware of any NoSQL databases that provide form support for constructing end-user interfaces that can feed directly into the database. eXist supports XForms (another W3C standard), which allows you to easily capture user input into XML documents in the database. Some organizations find that eXist is ideal not just for managing data collection with forms, but also for entire backend workflows around the content publishing process.

**Application development**

Like some RDBMSs and NoSQL databases, eXist is embeddable into your own applications. However, when you are using eXist as a server, it really becomes an application platform and offers more than almost any other database system.
When running eXist as a server, you can develop entire applications in eXist’s high-level query languages (XQuery and XSLT) without necessarily having to be a computer scientist or programmer.

Transaction management

Most RDBMSs and many NoSQL databases allow you to control your transactions from within your database queries or API calls. Unfortunately, eXist does not currently support database-level transaction control. eXist does have transactions internally and uses a database journal to ensure the durability and consistency of your data, but these are not exposed to the user. Transaction control is high on the list of desirable features for eXist, and some options have already been explored for the future.

Horizontal scalability and replication

A feature of many RDBMSs and NoSQL database systems is the ability to cluster database nodes to increase database scalability and capacity. eXist is currently mostly deployed on single servers. As of this writing, it has no support for automatic sharding of data. eXist has recently gained support for replication in version 2.1 through JMS (Java Message Service) message queueing. The replication feature allows you to have a master/slave database, which is highly available for reads that may occur from any node, but it is still currently recommended to send writes to the master node. For further details of the emerging replication support in eXist, see https://github.com/eXist-db/messaging-replication/wiki.

History

Once upon a time, around the turn of the 21st century, there was a researcher named Wolfgang Meier working at the Technical University of Darmstadt. He was in need of a system to analyze and query XML data, and since there was nothing around that satisfied his needs, he decided to write something himself: eXist.

Starting out in C++, Meier quickly turned to Java, and by the beginning of 2001, a first version was available. It was based on a relational database backend and, compared to where we are now, very primitive. The functionality was basic and it was slow on indexing, but yes, it already had some XPath on board. Immediately, some dictionary research projects started using it.

The next stage was replacing the relational backend with native XML storage. While this was happening, more and more people started using eXist, and around 2004 the first commercial projects arrived. The development of eXist has since then mostly been financed by its users, who needed new functionality and were willing to pay for it.

Implementing XQuery met some resistance. At that stage, eXist was still mostly an XML database only. Why would you need something like XQuery if you already have
XPath? Luckily (for us), a professor of literature really needed XQuery support and paid for its implementation. It was embedded in the product by 2005.

During 2005 eXist was going so well that Meier was able to quit his university job and concentrate on eXist projects only. By that time some other programmers had come on board, and they constitute what we now know as the original “core programmer team.” In alphabetical order, they were Pierrick Brihaye, Leif-Jöran Olsson, Adam Retter, and Dannes Wessels.

Up to 2006 the version number was kept to v0.xx, but in 2006 a real v1.0 was released!

By this time, having previously only communicated via the Internet, the core programmer team met live for the first time in 2007 in Versailles. One of the first things they did was to check eXist against the official XQuery test suite, which subsequently resulted in the current 99%+ compliance score.

The product kept evolving. A major improvement was replacing the existing scheme for node identifiers with a much better one. As a result of that, limitations on XML size and structure disappeared. Stability and transaction management were improved and the Lucene full-text search engine added. From the original research/retrieval tool, eXist evolved into something we really could call a native XML database.

The XRX (XForms, REST, XQuery) paradigm popped up as a way to create fully XML-driven applications. eXist was among the first engines that made this possible. It turned, slowly but surely, into a full-blown application platform.

With version 1.4 of eXist released in 2009, suddenly many more organizations were using eXist in their production systems day to day. More development effort went into stabilizing, fixing bugs, and improving reliability. With this, the first “settled application” problems arrived: it became harder and harder to change anything without breaking backward compatibility. Consider, for instance, eXist’s XQuery update support: an implementation of a draft version of the standard for writing XQuery statements to update XML. Switching to the final standardized version is virtually impossible because it would break backward compatibility and existing applications would stop working.

However, the development team did not stop working, and gradually the 2.0 version as we know it came to life. Release candidates were made available throughout 2012, containing a large number of major changes and additions to the previous versions:

- Behind the scenes, the XQuery engine and optimizer were improved.
- Support for (large parts of) XQuery 3.0 was added.
- The way the indexes work was redesigned to reduce lock contention, offer modularity, and improve performance.
• Security was reorganized and now works not only a lot faster, but also in a way most developers are comfortable with (i.e., similar to Unix-like systems).
• The repository manager was added, opening the way to a more modular eXist.
• RESTXQ, a standard for coupling function invocations to URLs, was added.
• And, of course, numerous other small improvements were made.

The final version 2.0, released in February 2013, was a massive leap forward from 1.4, representing the culmination of more than three years’ worth of sustained development effort. As such, it was not completely without backward compatibility problems. For instance, existing XQuery applications will have to do something about their security settings before they can run on the new version. However, that’s not too hard and is well worthwhile.

Version 2.1 was released shortly after, in July 2013, and consisted mostly of bug fixes and a new version of eXide. In February 2014, a release candidate of eXist 2.2 was made available, which—along with the usual bug fixes—included a completely new range index based on Lucene that offered much improved query times.

It is expected that eXist will keep evolving. The plans are to move more and more toward a core product with separate modules, enabling adding à la carte functionality as needed.

Competitors

Now, obviously we are passionate about eXist; otherwise, you would not be reading a book we have written on the subject. More importantly, though, we are passionate about open source, and even more so we are concerned with quality software and using the right tool for the job. Like any other product, eXist has both strengths and weaknesses, and it would be somewhat misleading if we were not to share the whole story with you. Pointing out the weaknesses of a software product for which you have bought a book may not help us sell more books, but we do hope it will help you make informed decisions.

As eXist has such a wide scope, it is impossible to compare it directly to other products; so, we compare it instead against other native XML databases that also couple web server and application platform capabilities.

eXist’s competitors can be split into two categories: those that are open source and freely available, and the closed source, commercial offerings. By no means is what follows a complete list, but it contains the offerings that we believe are popular and frequently encounter when talking to others.

A further independent comparison is available in the XML database article on Wikipedia.
Open Source Competitors

Let’s begin with the open source eXist competitors.

**BaseX**

BaseX is a lightweight native XML database with some application server facilities written in Java. The project was started in 2005 by Christian Grün at the University of Konstanz, and BaseX was released as open source in 2007. BaseX promotes ease of use and provides an easy-to-use GUI frontend also written in Java.

Compared to eXist, BaseX adheres more closely to the W3C XQuery specifications, achieving 99.9% compliance with the W3C XQuery 1.0 specification (eXist has 99.4%) and implementing the specifications for XQuery Update 1.0 and XQuery Full-Text. eXist has an older draft implementation of XQuery Update and its own proprietary full-text search. eXist, however, has been available for significantly longer, and thus benefits from many more features, such as XSLT.

BaseX is released under the more liberal BSD license. Commercial support is available for BaseX from BaseX GmbH, which was founded to support commercial applications of BaseX.

**Sedna**

Sedna is a lightweight native XML database without application server capability, written in C and C++. The origins of Sedna are not well documented, but it appears to have started around 2003 as a project of the Institute for System Programming at the Russian Academy of Sciences. Sedna seems to focus on providing core database services and little more. While it has no REST Server of its own, it can be configured to work as a module within the Apache HTTP Server.

Compared to eXist, Sedna supports more APIs for different programming languages directly; eXist mostly assumes that developers will use its REST or RPC APIs, and leaves language APIs to third-party providers. Sedna reports 98.8% compliance with the W3C XQuery 1.0 specification; as mentioned previously, eXist has 99.4% compliance. Sedna, like eXist, implements its own proprietary full-text search, and a draft version of XQuery Update.

Sedna is released under the Apache 2.0 license. There does not appear to be a commercial support offering for Sedna.

Closed Source, Commercial Competitors

Now we’ll take a look at eXist competitors that are closed source and commercially available.
28.io

28.io is a PaaS (Platform as a Service) for the Zorba open source XQuery processor. 28.io integrates Zorba with MongoDB, supporting the storage and indexing of XML into MongoDB as its main datastore. The query optimizer leverages the full capabilities of MQL (Mongo Query Language), enabling developers to leverage the expressiveness and productivity of XQuery atop a highly scalable store.

In comparison with eXist, 28.io focuses on the cloud and manages its database entirely using XQuery, whereas eXist provides a number of Admin GUI tools and additional APIs. 28.io, much like eXist, provides an application server platform enabling you to build entire apps using XQuery. 28.io (through Zorba) has similar compliance to the W3C XQuery 1.0 specification as eXist, but also supports XQuery Update, XQuery Full-Text, and XQuery Scripting. 28.io’s main advantage over eXist is its cloud scaling; eXist’s main advantage is XSLT, XRX, and XForms support.

28.io is developed by 28msec. 28msec is based in Zurich, Switzerland, and has strong research links with ETH Zürich.

MarkLogic Server

MarkLogic Server is a standalone native XML database server, written in C++, that provides XQuery and XSLT query and transform capabilities. MarkLogic Server also has the capability to cluster nodes to scale horizontally, with the additional capability to pass large batch processing jobs off to Hadoop. MarkLogic distances itself from the technical marketing of XML and XQuery and instead identifies itself as a NoSQL database solution for the enterprise.

Compared to eXist, MarkLogic markets itself as being able to handle petabytes of XML data. eXist can currently scale to hundreds of gigabytes, but this is very much dependent on the dataset and queries made. MarkLogic lacks a document-representative REST API, but does provide a REST API for application development. MarkLogic’s main advantage over eXist is scaling to huge datasets, while eXist’s advantage is its fast innovation and rich feature set. Both support XSLT, but MarkLogic does not support XQuery Update; rather, it provides its own proprietary functions.

MarkLogic Server is developed by MarkLogic Corporation, based in San Carlos, California.

Who Is Using eXist, and for What?

The problem with giving something away for free with no questions asked is that you can never quite be sure:

- How many people are using it
Who the people using it are
What it is being used for

From the support channels available to eXist users, and as a member of the community, you can see that eXist is used by many people for many different purposes, but their end goals and projects are not always disclosed or clear.

Here we have pulled together a few descriptions of various projects using eXist from the developers of those projects themselves:

The Tibetan Buddhist Resource Center (TBRC) holds the world’s single largest collection of Tibetan texts—nearly 10 million scanned pages and over 11,000 Unicode Tibetan texts. TBRC.org provides online access to over 4,000 users via an Ajax client written in Google Web Toolkit as a front-end to the eXist-db. TBRC has used eXist-db since 2004 to store the catalog for the texts in the library as well as a knowledge-base of persons and places that provide a cultural context for Tibetan literature. The integration of eXist-db with the Lucene full-text indexing has created a powerful framework with which TBRC.org is able to provide searchable access to the library via comprehensive tables of contents in Tibetan and a large collection of texts that have been input in Unicode in centers around the world. Our production system currently runs eXist-db 2.1.

—Chris Tomlinson,  
Senior Technical Staff Member,  
Tibetan Buddhist Resource Center,  
Cambridge, Massachusetts

ScoutDragon initially started as a baseball research project by a group of baseball enthusiasts including writers, agents, scouts, fans, fantasy owners, and even former players. This group realized a need for original English content, data, and research on baseball players in Asia.

All data for multiple sports covering multiple sporting leagues is stored in XML documents within eXist in a schema derived from IPTC’s SportsML, most extensions having to do with providing multi-lingual support of players so that information may be displayed in English, Japanese, Korean, and/or Chinese. XQuery has proven to be a fantastic language for not just transforming the vast quantities of data to web pages, but also for data analysis and the generation of sabermetrics-based statistics.

—Michael Westbay,  
Lead Programmer/System Administrator,  
ScoutDragon.com,  
Japan

Semanta’s core business is metadata in business intelligence. Part of our concern is parsing metadata from reporting platforms. Many of these reporting platforms supply their metadata in large XML chunks, which we then need to further process efficiently. A typical example is our IBM Cognos connector, where we use eXist heavily to extract details of report structures and data sources. Originally we thought we would only use eXist for prototyping, but ultimately, we have used embedded eXist in the production
system; re-writing the connector without eXist’s XQuery turned out to be just too complicated!

—David Voňka,
Programmer,
Semanta,
Czech Republic

The Centre for Document Studies and Scholarly Editing of the Royal Academy of Dutch Language and Literature (Ghent, Belgium) develops rich scholarly collections of textual data, and publishes them as digital text editions and language corpora. From the start, we have fully embraced open standards and publication technologies. At first, we started out with the Cocoon XML publication framework, which back then nicely integrated with eXist (or the other way round) for efficient querying of XML content. Since the introduction of eXist’s MVC framework, we have extended our use of eXist as a full application server, not only for querying the indexed data, but also driving the entire application and presentation logic.

The texts we’re querying (or rather, processing) with eXist are mostly document-centered XML documents that are conformant to the schemas developed by the Text Encoding Initiative (TEI). Depending on the specific edition project, they are enriched with metadata such as named entities, editorial annotations, and sometimes highly specific textual documentation (such as critical apparatuses documenting variation among text versions). Though our texts are mostly in Dutch, we try to connect and contribute to methodological good practice emerging in the interesting field that is Digital Humanities. Some of our exemplar projects include a collection of letters in relation to the Belgian literary journal *Van Nu en Straks*; a digital edition comparing 20 versions of *De trein der traagheid*, a novel by the Belgian novelist Johan Daisne; and a digital edition of the first Dutch dialect survey in the Flemish region by Pieter Willems (developed between 1885–1890).

—Ron Van den Branden,
Centre for Scholarly Editing and Document Studies of the Royal Academy of Dutch Language and Literature, Ghent, Belgium

At the Cluster of Excellence “Asia and Europe in a Global Context,” we use eXist-db to store our collections of MODS (bibliographical) and VRA (image metadata) records. We have developed two open source applications for this, Tamboti and Ziziphus, where our records can be searched and edited. Both applications are built entirely in XML technologies (XQuery and XForms) using eXist-db and make use of LDAP integration and detailed user rights management.

—Heidelberg Research Architecture, Cluster of Excellence “Asia and Europe in a Global Context,” The University of Heidelberg, Heidelberg, Germany

Haptix Games is a video game and interactive application development and publishing studio, and we have been a Microsoft shop for as long as I can remember. We have
leveraged C#, MVC, and IIS for user experience; WCF, OData, and BizTalk for message exchanges; MSSQL and Entity Framework for storage. That is a lot of acronyms and even more complexity under the hood. Prototyping a concept usually involved all above-mentioned technologies, while the final solution release was either expensive, inflexible, or did not meet client expectations.

With the adoption of eXist-db our development and release workflows have become highly agile and more competitive. Utilizing eXist-db as a dark-data solution platform and not just another XML database allowed us to eliminate 80% of our Microsoft code base just by taking advantage of the built-in web server, low-level data manipulation using XQuery 3.0, restful data exchange, and native storage capabilities.

—Chris Misztur,
CTO,
Haptix Games,
Chicago, Illinois

easyDITA is an end-to-end solution for collaboratively authoring, managing, and publishing content using the DITA XML standard. Companies utilize easyDITA to reduce the cost and time to market to deliver content in a variety of formats and languages. By leveraging eXist, easyDITA is able to deliver customers exceptional ability to search, manage, localize, and publish content. eXist’s schemaless design and flexible indexing system makes it easy to support customizations like reporting, analytics, and new content models without sacrificing performance or doing major redesigns.

—Casey Jordan,
cofounder,
easyDITA, Jorsek LLC,
Rochester, New York

We [at XML Team Solutions] help media and entertainment companies integrate sports news and data feeds. These feeds are predominantly XML. We use eXist for two things:

1. Regulating and preparing vendor web service XML for transmission to clients. Scheduled jobs access remote web services, preprocess, and pass on XML via HTTP Client to our main feed processor.

2. API to drive graphics for live television broadcast. API built from RESTXQ provides live updates of results to broadcaster clients. Currently uses JMS to sync from one write DB to two load-balanced readers. Also has an XForms “beat the feed” live score updater which mimics the incoming feed in case feed vendor is delayed.

We recently delivered a project for BBC Sports to deliver live broadcast information. eXist met all the requirements for speed, cost, and reliability for an API to deliver up-to-date scores and statistics to BBC television.

—Paul Kelly,
Director of Software Development,
XML Team Solutions Corp,
Canada
eXist-db is at the core of [the Office of the Historian’s] open government and digital history initiatives. It powers our public website, allowing visitors to search and browse instantly through nearly a hundred thousand archival government documents. On the fly, it transforms our XML documents and query results into web pages, PDFs, ebooks, and APIs and data feeds. Its support of the high-level XQuery programming language and its elegant suite of development tools empower me and my fellow historians to analyze data and answer research questions.

The open source nature of eXist-db has delivered far more value to us than its simply being “free”; its active, welcoming, expert collaborative user community has helped us learn, discover eXist-db’s plethora of capabilities, and find the best solutions to our research and publishing challenges. eXist-db belongs in the toolkit of all digital humanities, open government, and publishing projects.

—Joe Wicentowski, Historian, Office of the Historian, U.S. Department of State

Contributing to the Community

There is a vibrant and supportive community around the eXist software, whose goal it is to make using eXist easy for beginners and as painless as possible for advanced developers. The eXist community prides itself on the agility and quality of its responses to support requests.

There are many ways to contribute to eXist and the community. You need not be a crack software engineer; even beginners asking questions on the mailing list can help others learn from their issues and encourage the developers to simplify or consider new approaches.

To get in touch with the eXist community, you have several channels available to you:

Email: the eXist-open mailing list
This is the official preferred mechanism, and your best bet for getting a quick answer. There are also the eXist-development and eXist-commits mailing lists; the former is used for technical discussion of features and fixes that go into eXist, and the latter is a feed of any changes made to the source code of eXist.


Stack Overflow: the exist-db tag
While the mailing lists should currently be considered the primary support mechanism, Stack Overflow is also becoming popular for asking eXist questions. You can find eXist questions and answers under the exist-db tag.
Twitter: @existdb

The Twitter channel is monitored by the core developers of eXist. It’s mostly used for announcements about eXist, as providing tech support in 140 characters is tough!

IRC: #existdb on irc.freenode.net

The eXist IRC chat room is for the community, by the community. It can be a mix of users and developers and is often worth a visit, but getting a response can really depend on who is awake and logged in.

Individuals Using eXist

As a user of eXist, be sure to do the following:

Ask questions

There are no stupid questions. Importantly, all questions and answers on the mailing list are archived so that others may learn from them also. Sensibly, you should search the archive first, to see if your question has already been answered; if not, or if the answer’s not clear, then ask away!

Report bugs

All software has bugs! eXist is no exception. If you think you have found a bug, it is probably best to discuss it on the eXist-open mailing list first, and then, if it’s confirmed, log it in the eXist issue tracker on GitHub.

If you want to report a bug, it’s very important that the developers of eXist understand how to reproduce what you are seeing; if you can’t describe the problem and its cause, it’s very hard for the community to help you! Ideally you should provide a reproducible test case of the absolute minimum steps required to cause the issue. See “Getting Support” on page 413 for more detail.

Answer questions

As you begin to use eXist, you will start to learn more and more things that other users may not know. Why not get some good karma back by answering some questions on the mailing list? After all, it’s a community!

Evangelize

If you’re having a great time using eXist, or you are enthralled by some neat feature, tell your friends, and let us and everyone else know by writing a blog entry or article.

Organizations Using eXist

Open source developers are often working on a project for “the love of it,” and many of the developers of eXist contribute much of their time to the project completely unpaid. Sadly, love for developing open source code with your friends does not nec-
essarily equate to food or shelter. If you’re part of an organization making free use of eXist, there are a number of ways that you can contribute back to the community:

**Sponsor features or bug fixes**
Perhaps there is some feature that you wish that eXist had that would really help your project, or there is a bug that sometimes upsets your system. Your organization could financially sponsor a developer from the eXist community to add this feature or resolve that issue. Sponsoring eXist developers for small or large projects helps support them in their work on eXist and could provide new or improved functionality to the community. If you want to give something back financially but have no specific features or bug fixes in mind, just get in touch via eXist-open and the community will helpfully propose a project to meet your budget.

**Friday afternoon eXist**
If you have developers in your organization, empowering them to spend a small amount of their paid work time contributing to the development of eXist can also be a great way to give back to the community. For example, Google allows its developers to work on open source projects on Friday afternoons and has realized various benefits from this.

**Contracts and jobs**
Are you looking for someone who is an expert in eXist and XQuery and/or XML technologies? The eXist-open mailing list can be a great place to advertise. You will more than likely end up sponsoring one of the contributors to eXist, as they tend to be the people who really know it inside and out.

**Support and maintenance contracts**
If you’re serious about using eXist in your projects and running production systems on it, you will more than likely want the support and operational security afforded by purchasing a support contract for it. eXist Solutions provides a variety of support contracts and consultancy services for eXist. It was founded by core developers of eXist and contributes almost all of its resources back into developing the software. By working with eXist Solutions, you are closely supporting and funding eXist’s development.

**Authors Using eXist**
If you’re an author using eXist, here’s how you can contribute:

**Documentation**
eXist has a large set of documentation that accompanies it, but it is by no means complete or exhaustive. You do not have to be a developer to write documentation for eXist, and all improvements to the documentation are warmly accepted.
Developers Using eXist

Developers using eXist can give back in the following ways:

Bugs, patches, and new features

Found a bug? Want to submit a patch or new feature? Why not roll up your sleeves and get your hands dirty? In the beginning the eXist code base may seem intimidating in its size, but it’s fairly modular and easy to get around. And if you have the skills, there is often no quicker way to get something fixed than to do it yourself, while hopefully learning a few new and interesting things along the way. Bug reports should be posted to the eXist-development mailing list first, and then logged in the issue tracker on GitHub. Patches can be submitted by means of a pull request to the eXist GitHub repository.

For further information about developing eXist, see “Developing eXist” on page 483.

Additional Resources

This section contains additional informational resources. It’s compiled from our personal preferences and bookshelves, meaning there are many other good sources of information around. However, this list is a good place to start:

General

- **W3C (World Wide Web Consortium)**
  The W3C is the body that manages, among other things, the XML standards. The website is surprisingly easy to use, yet informative.

- **W3 Schools**
  For a quick high-level overview of any W3C standard with practical examples, try the W3 Schools.

XQuery

- **XQuery**, by Priscilla Walmsley (O’Reilly, 2007)
  This is probably the best XQuery book available in our opinion.

- **XQuery wikibook**, edited by Dan McCreary et al.
  The XQuery wikibook is an excellent resource for XQuery and eXist, with the majority of the examples developed for eXist.

- **XRX wikibook**, edited by Dan McCreary et al.
  The XRX wikibook, like the XQuery wikibook, is an excellent resource when you’re building applications atop eXist using REST and XForms.
• **XPath and XQuery Functions and Operators 3.0 (W3C, 2014)**
  The F+O specification is a great resource for quickly looking up the available functions and their specification for XQuery, XPath, and even XSLT.

• **XQuery: The XML Query Language**, by Michael Brundage (Addison-Wesley, 2004)
  This was a great book at the time it was published; while still relevant, it was released before the final XQuery 1.0 specification.

• **XQuery from the Experts: A Guide to the W3C XML Query Language**, by Don Chamberlin et al. (Addison-Wesley, 2004)
  Again, this book was released before the final XQuery 1.0 specification, but it is useful for those who want to know the nitty-gritty details like the formal underpinnings of the language.

• **The XQuery Talk mailing list**
  The xquery-talk mailing list is a great place to ask XQuery questions that are not specific to eXist-db.

---

**XSLT**

  This is the book to have beside your keyboard if you ever want to do any serious XSLT programming.

• **XSL-List**
  The XSL-List is the best place to ask XSL questions and receive help. Note that it has an excellent archive; we suggest that you search that first for an answer before asking!

• **XSLT Questions and Answers—FAQ**, curated by Dave Pawson
  The XSLT FAQ is an incredible resource that has many answers from those who were involved in specifying XSLT and those recognized as subject experts.


  Also a very handy book to have when you only sporadically program XSLT; it contains many useful “recipes.”

---

**XForms**

• **XForms Tutorial and Cookbook wikibook**, edited by Dan McCreary
The XForms wikibook is an excellent resource for XForms examples, especially as many of the articles are developed against eXist.

- **XForms Essentials**, by Micah Dubinko (O’Reilly, 2003)
  
  An excellent reference guide to have when you’re working with XForms, with some good explanations of the W3C XForms specification.

**XML Schema**

- **Definitive XML Schema**, by Priscilla Walmsley (Prentice Hall, 2013)
- **RELAX NG**, by Eric van der Vlist (O’Reilly, 2003)

**XSL-FO**

- **XSL, XSL-FO FAQ**, curated by Dave Pawson
  
  The XSL-FO FAQ is in a similar vein to the XSLT FAQ and likewise is an invaluable resource, with many questions answered by subject experts.
This chapter takes you through the first steps in using eXist. It handles subjects like downloading and installing, starting and stopping, running the examples, and demonstrates some of eXist’s capabilities on a “Hello World” level. In other words, like the chapter title says, it will get you started.

If you have used eXist before, you may like to skip over this chapter.

**Downloading and Installing eXist**

This section takes you through the steps necessary for getting eXist up and running on a standalone development system. That is to say, we keep things simple and don’t spend time on more advanced subjects such as database security, tuning, performance, embedded mode operation, and the like. Those subjects and more are covered in the chapters to come.

Be aware that installing eXist for production purposes (e.g., as the engine behind a public website) requires much more thought and planning. Security, especially, requires attention in those kinds of more public situations. Also, if you plan to use eXist with some really huge datasets, you probably need a different setup than that described here. For information on installing eXist in a server environment, see “Installing eXist as a Service” on page 405.

**Preconditions**

eXist can be installed on almost all versions of Linux, Windows, and Mac OS X. The deciding factor is whether or not your OS (operating system) supports at least Java version 1.6 (1.7 is recommended). If it does, then eXist should run.
In order to run the eXist installer, you must have a working JRE (Java Runtime Environment) or JDK (Java Development Kit), version 1.6 or newer. The eXist team regularly tests eXist with the Oracle and OpenJDK JRE and JDKs, but the community reports that the IBM JDK (among others) also works.


To check whether you have the right Java version (and have installed it correctly), open a terminal/command-line window and type `java -version`. You should see a message telling you which version of Java you’re running.

**Downloading eXist**

Downloading eXist is easy. Go to http://www.exist-db.org, navigate to the download section, pick the right distribution, and download it. For getting started, pick the latest stable distribution. The filename will probably look like `eXist-db-setup-<version>-rev<XXXXX>.jar`.

This book was based on the 2.1 release of eXist (`eXist-db-setup-2.1-rev18721.jar`), but by the time you read this, a newer version may be available.

**Things to Decide Before Installing**

Of course, you can go ahead now and run the installer using the defaults provided. However, there are probably a number of things you want to decide before firing up the installer:

*Installation directory*

Where are you going to install eXist? For a “getting started” installation, this is not extremely important; you can use the default suggested by the installer or any other location you like (provided it is writable by the installer).

However, there are a number of reasons why the installation directory matters more than is usual for a software installation. Firstly, the default for the data directory (where eXist stores its data) is inside the installation directory, as described shortly. Secondly, logging and temporary directories are also inside the installation directory.

Having frequently written locations inside a software installation might be problematic because security sometimes does not allow this, or it causes performance...
degradation. When you start to do more serious work, make sure the important locations are included in your backup.

We will refer to the installation directory as $EXIST_HOME throughout this book.

**Data directory**

This is the directory where eXist stores the content of its database. The installer will propose a default that’s inside eXist’s installation directory ($EXIST_HOME/webapp/WEB-INF/data, to be precise). If you just plan to play around a bit or do some development work, keep the default. You can always change it later.

However, if things get serious, like on a production server, make sure that this directory is writable, located on a volume that is sufficiently fast for updates, and backed up (which is not always the case for program file directories).

**Administrator password**

The installer will ask you to provide an administrator password. This is not your operating system’s administrator password, but the initial password used for eXist’s administrator’s account (called admin). You are strongly encouraged to set an administrator password on all installations of eXist. If you don’t, eXist will use an empty password, so anyone who tries an empty password would have full access to your eXist instance.

**Memory settings**

The installer allows you to set the amount of memory reserved for eXist’s JVM and its internal cache. Common settings are shown in Table 2-1.

<table>
<thead>
<tr>
<th>Max memory</th>
<th>Cache memory</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>512 MB</td>
<td>64 MB</td>
<td>Don’t go any lower than this, or eXist will not run properly.</td>
</tr>
<tr>
<td>1,024 MB</td>
<td>128 MB</td>
<td>This is the default setting and is fine for small development use.</td>
</tr>
<tr>
<td>2,048 MB</td>
<td>256 MB</td>
<td>If your machine has enough memory to spare, then use at least this.</td>
</tr>
</tbody>
</table>

**Packages/apps to install**

For getting started purposes, we recommend keeping everything checked (this book assumes that you did!).

---

**Download and Installing eXist** | 21
Installing eXist

Start the installer in one of the following ways:

For desktop-driven systems
   If Java is set up correctly, on many systems that provide a GUI, double-clicking the downloaded eXist-db-setup-<version>-rev<XXXXX>.jar file will fire up the graphical installer.

On all GUI systems, from the command line
   Open a terminal/command-line window and enter the following command: java -jar eXist-db-setup-<version>-rev<XXXXX>.jar (of course, the name of the file you just downloaded). This will launch the graphical installer.

On non-GUI systems, from the command line
   If you are on a system that does not provide a GUI environment—for example, a remote server—you can entirely install eXist from the terminal. At the terminal/command-line window, enter the following command: java -jar eXist-db-setup-<version>-rev<XXXXX>.jar -console (using the name of the file you just downloaded).

As usual with installers, follow the instructions on the screen to complete the installation. You’ll be asked to enter the information prepared in the previous section. Let the installer run its course, and that’s it!

Post-Installation Checks

By default, eXist uses two TCP ports:

Port 8080
   This port is used for all the normal HTTP communication.

Port 8443
   This port is used for the confidential HTTPS communication. By default, eXist uses a self-signed certificate, which, while more secure than using no certificate, should not be considered for production use. You may also see a warning about the self-signed certificate when accessing this from a web browser.

If one of these ports is used by another application on your system, you either have to make this other application change its ports or change the port settings for eXist.

The easiest way to find out if something is using these ports is, before starting eXist, to visit http://localhost:8080/ and https://localhost:8443/. If nothing happens, the ports are probably free and you can go ahead.
Changing eXist’s TCP port usage is explained in “Changing Jetty Settings: Port Number and URL Prefix” on page 205.

This book assumes eXist is running on localhost using the standard TCP port numbers 8080 and 8443, so you’ll see URLs like http://localhost:8080/… throughout the book.

Starting and Stopping eXist with a GUI

If you’re on a system with a GUI, the installer will have created a menu entry and/or a desktop icon called eXist database. If you’re on a command-line-only system, go to “Starting and Stopping eXist from the Command Line” on page 24.

Clicking the eXist database icon starts eXist and also fires up a little control application that should be visible in the system tray (or its equivalent on your system) as a dotted X. For instance, on a Windows 7 machine it looks like Figure 2-1.

![Figure 2-1. The eXist control application in a Windows 7 system tray](image)

Clicking it opens a little menu that gives you further control of eXist (like stopping the server) and lets you do a few other useful things, as shown in Figure 2-2.

![Figure 2-2. The menu of the eXist control application](image)

If for some reason this doesn’t work, open a command window in $EXIST_HOME and type java -jar start.jar. This should fire up the control application and the database. If this works, you’re probably best off creating a shortcut or menu entry for it manually. If still nothing happens, read the next section.
After starting the database, open your browser and visit http://localhost:8080/exist. If a nicely tiled screen appears (like in “The Dashboard” on page 24), you’ve succeeded!

**Starting and Stopping eXist from the Command Line**

If you don’t want to or can’t work with the GUI niceties, you can also start eXist from the command line. For this, open up a command-line window and navigate to $EXIST_HOME/bin. There you’ll find several command files in both the Windows (*.bat versions) and Unix/Linux/Mac (*.sh versions) variants. For starting and stopping, do the following:

```
startup
```

This will fire up eXist.

```
shutdown -p adminpassword
```

This will stop the running eXist instance. It needs the administrator password.

**A First Tour Around Town**

This section will give you a quick tour of eXist’s highlights and attractions, including the user interface and what’s on your disk.

**The Dashboard**

The home screen of eXist since 2.1, http://localhost:8080/exist, is called the dashboard; it is a set of tiles linking to various applications and utilities. The initial set shows the default tiles provided with eXist. You may install additional ones via the Package Manager, or if you start developing applications of your own with the Packaging System (see “Packaging” on page 227), those can appear here too.

Now, most of the functionality provided through the dashboard—stuff like eXide and the function documentation—is important, and you will probably use it often. It is therefore well worth your time to familiarize yourself with the smorgasbord offered (see Figure 2-3).
The tiles provided by default are:

**Java Admin Client**

This tile provides a Java Web Start, a.k.a. JNLP (Java Network Launching Protocol), link to eXist’s Java Admin Client application. Use this if you want to access an eXist installation remotely, from a system that does not have eXist installed. For local use you’re better off starting the Java Admin Client directly (e.g., through the control application’s menu, as shown in Figure 2-2). Read more about the Java Admin Client in “The Java Admin Client” on page 29.
JNLP does not work well with all browsers. You might just get a “Save downloaded file” dialog when pressing this tile.

Admin Web Application
This tile gives access to the original (pre-2.x) administrator web client application. There is still some functionality there that has not yet appeared in the new interface, such as profiling queries and index overview.

Collections
This tile starts a collection browser that enables you to control the contents of the database.

Shutdown, Backup
These applications do what their titles suggest.

Package Manager
A package is a set of related files that together provide some kind of functionality—for instance, an application or library. The Package Manager allows you to manage (view, install, and uninstall) packages in your eXist database. When you open it, you can see that most of the functionality behind the tiles of the dashboard is provided by separate packages.

Packages can come from the eXist public repository—you can see the packages available there by selecting the available option at the top of the Package Manager—or they can be distributed as separate package files with the extension .xar.

It is also possible (and even advised!) to design your own applications for use with the Package Manager and distribute them using .xar package files. You’ll learn how to do this in “Packaging” on page 227.

User Manager
This tile allows you to control the user population of the eXist database. You can create, edit, and delete users and groups.

betterFORM Feature Explorer, betterFORM Demos, XSLTForms Demo
eXist has two built-in ways of doing XForms: betterFORM™ and XSLTForms™. These applications provide you with demos and overviews. Find more information in “XForms” on page 254.

eXist-db Demo Apps
This tile is a collection of applications that demonstrate some of eXist’s capabilities.
XQuery Function Documentation
This is an application you’ll probably use very often. It provides an overview of all the functions available and their documentation in both the standard eXist extension modules and your own XQuery modules. There’s more about modules in Chapter 7.

eXist-db Documentation App
This app provides access to the eXist documentation.

eXide
eXide is a cool, handy, fully integrated editor for working with XQuery, XML, and other resources stored in eXist. You can use it for a multitude of activities, from writing complete applications to fiddling around and experimenting. Don’t miss it. Find more information in “eXide” on page 374.

Playing Around
If you’re like us, at this point you’ll want to play around, try some XQuery, store some XML, and perform other familiarization rituals. Get your feet wet and splash around (without going into the deep end). Here is the quick recipe:

2. Click on the eXide tile.
3. Click Login and log in as admin with the password set during installation.
4. Directly type some XQuery and run it.
5. If you want to see what’s in the database, click File→Manage (or press Ctrl-Shift-M).
6. If you want to save your work or put some related files together, create a collection for this underneath /db.

What’s in Your Database
You can look inside the database using, among others, the Collections app in the dashboard or the eXist Java Admin Client (see “The Java Admin Client” on page 29). You’ll see something that looks like a disk directory structure (but of course isn’t). To explain the terminology, what you might think of inside your database as a directory is called a collection in XML database geek speak (more about this in “Terminology” on page 88). Here are the most important collections:

/db
The root collection in the database is always /db. You can’t change this.
/db/system
This is where eXist stores important configuration information (e.g., about users, groups, and versioning). You shouldn’t change any of this information by hand or programmatically, with the exception of what’s inside /db/system/config.

/db/system/config
This collection is used to store the collection-specific configuration for eXist, like validation, indexes, and triggers. If you look underneath, you’ll find a (partial) copy of the existing database structure with collection.xconf files here and there. These (XML) files contain the collection configuration. Read more about this in “Implicit Validation” on page 246 and “Configuring Indexes” on page 275.

/db/apps/*
These are the root collections for the packages, installed during installation and manually later. Underneath these is their code and data. If you’re ever going to write applications yourself (Chapter 9), you’ll create your own subcollections here.

**What’s on Your Disk**

Now let’s look at some interesting and/or important locations on your disk for eXist.

There are rumors on the grapevine that the basic file structure will change in future versions, so be aware if you use this book with a later version than 2.1.

$EXIST_HOME/
This is eXist’s home directory.

$EXIST_HOME/conf.xml
This is eXist’s main configuration file. If you peek inside (it’s well documented), you’ll find entries for, for instance, all kinds of default behavior, the location of the database (in db-connection/@files), cache sizes, the indexer, and the built-in XQuery modules.

$EXIST_HOME/tools/jetty/etc/jetty.xml
This is the Jetty web server’s configuration file (eXist uses Jetty to communicate with the world). There are several interesting things you might want to change using this file, like the TCP port numbers and the default URL prefix exist/.

$EXIST_HOME/webapp/WEB-INF/
This location defines the eXist web application. It holds several important configuration files and is the default base location for the database and the logfiles.
$EXIST_HOME/webapp/WEB-INF/controller-config.xml
This tells eXist what to do when a request with a certain URL is entered. There’s more information in “The controller-config.xml Configuration File” on page 206.

$EXIST_HOME/webapp/WEB-INF/data/
This is the default location for eXist’s database (unless you specified somewhere else during the installation process).

If you peek inside this directory, you’ll find underneath the *fs subdirectory all the non-XML files stored in the database. However, your XML files are not there; they have seemingly disappeared. Don’t despair: they’re absorbed into the *.dbx files you see in the root of the database directory. You’ll find more information about this in “Help: Where Is My XML?” on page 87.

You might be tempted to change the non-XML content underneath the *fs subdirectory directly. Don’t do this. It will ruin the database’s internal administration. Use only the normal mechanisms for this, like WebDAV, the dashboard, or the Java Admin Client tool.

$EXIST_HOME/webapp/WEB-INF/logs/
Here you’ll find several logfiles that can help you find out what’s going on underneath eXist’s hood.

The Java Admin Client
Through the eXist controller application (visible in the system tray), you can start the Java Admin Client. This pops up a small and, admittedly, rather old-fashioned-looking program. It allows you to do maintenance work on the database like backups and restores, imports and exports, checking and setting properties, and creating collections. Figure 2-4 shows how it looks on a freshly installed database.

The eXist Client tool is a standard GUI application, and its functionality speaks for itself.

Most of this tool’s functionality is also present in the new dashboard application, so there’s a good chance you’ll never need it. However, there are circumstances in which it can be useful, such as when you’re working on a production server where you don’t want the dashboard to be present.
Getting Files into and out of the Database

eXist is an XML database. Its primary storage concern is XML documents. It can also hold your XQuery files, and any other resources needed by your application. So how do you get files in and out of it?

Collections app

Browse to eXist’s Collections application (available through the dashboard). This allows you to look through the contents of your database and maintain the collections and resources.

eXide

eXist’s built-in native IDE, called eXide, has facilities for uploading and downloading files. Click File→Manage (or press Ctrl-Shift-M).

WebDAV

eXist’s WebDAV (Web-based Distributed Authoring and Versioning protocol) interface allows you to access the contents of the database just like it was any other file store available to your OS. The address to use is http://localhost:8080/ exist/webdav/db/ or, when your OS requires safe URLs (like Windows 7), https://localhost:8443/exist/webdav/db/.
Exactly how to work with a WebDAV server and which client tool to use is platform-specific. Some operating systems, like Windows, will allow you to integrate it more or less into the normal file browsing capabilities, while others need special client tools. Read more about this in “WebDAV” on page 305.

**Java Admin Client**

eXist’s Java Admin Client tool (see “The Java Admin Client” on page 29) also has some basic facilities for getting files into and out of the database.

**External IDE**

Some external IDEs, such as oXygen, provide you with the option to work with eXist natively. This includes importing/exporting files. Find more information in “oXygen” on page 375.

**Programmatically**

Of course, you can import and export files programmatically by writing some XQuery code that performs what you want on the database. That’s okay within applications, but a bit cumbersome for now. For more information, see “Controlling the Database from Code” on page 107 and Chapter 13.

**Ant**

eXist provides a library for the Ant build tool to automate common tasks like backup/restore or importing a bunch of files. This method is recommended if you need to repeat batch tasks on your database. There’s more information in “Ant and eXist” on page 379.

**Hello eXist!**

This section performs a first exploration of the fundamental mechanisms in eXist; that is, how you get it to actually do something—store/retrieve/filter information, show a web page, transform XML, and more. In other words, this section is an extended “Hello world” example in which, in a (very) shallow way, we touch upon the important processing features of the platform.

For most of the examples, the output is not shown because we want to encourage you to try this yourselves using the provided example code. We assume you’ve installed the example code and know how to access it, as described in “Accompanying Source Code” on page 15.

**Hello Data**

In the example code for this book, there is an XML file in /db/apps/exist-book/getting-started/xml-example.xml that looks like Example 2-1.
Example 2-1. XML file

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Items>
    <Item name="Bogus item">This is a complete bogus item</Item>
    <Item name="Funny item">Ha, ha, very funny indeed!</Item>
</Items>
```

Accessing data (and also scripts) is done through the eXist REST interface. To see it in action, fire up your browser and visit `http://localhost:8080/exist/rest/apps/exist-book/getting-started/xml-example.xml`.

The result is that you see exactly the file from Example 2-1. Not impressive, maybe, but hey, this is only the beginning.

The REST interface allows you to directly query this file. For instance, assume you’re interested in the first item only. You can access it by adding a `_query` parameter:

```http
```

The result will be:

```xml
<exist:result xmlns:exist="http://exist.sourceforge.net/NS/exist" exist:hits="1" exist:start="1" exist:count="1">
    <Item name="Bogus item">This is a complete bogus item</Item>
</exist:result>
```

Because it’s a query, eXist wraps the result in an `exist:result` element with additional information in its attributes. There are other query parameters that will let you limit the size of the result set and even retrieve the results block by block. More information about the REST interface can be found in “Querying the Database Using REST” on page 94.

Hello XQuery

Of course, the main language when you are dealing with eXist is XQuery, which is the language to access XML databases. Put your XQuery script in a file (or database document) with the extension `.xq`. Example 2-2 shows you a basic way to output some XML.

Example 2-2. Basic XQuery code returning XML

```xquery
xquery version "3.0";
let $msg := 'Hello XQuery'
return
    <results timestamp="{current-dateTime()}">
        <message>{$msg}</message>
    </results>
```
Now what if you want to show the result as an HTML page instead? That’s called serialization, and Example 2-3 shows one of the ways to do it.

Example 2-3. Basic XQuery code returning HTML

```xquery
xquery version "3.0";

declare option exist:serialize "method=html media-type=text/html";

let $msg := 'Hello XQuery'
return
  <html>
  <head>
    <title>Hello XQuery</title>
  </head>
  <body>
    <h3>It is now {current-dateTime()} and so {$msg}!</h3>
  </body>
</html>
```

XQuery-initiated readers might have noticed that we did not declare the exist namespace prefix. eXist has most eXist-specific namespace prefixes predeclared for you, so you don’t have to explicitly mention them in your code.

**Hello XSLT**

XSLT is built into eXist using (by default) the Saxon XSLT processor. The examples contain a simple stylesheet to show you how this works. The stylesheet in Example 2-4 takes the XML from Example 2-1 and turns it into an HTML page.

Example 2-4. Transformation of the example XML into HTML

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:template match="/">
    <html>
      <head>
        <title>Hello XSLT</title>
      </head>
      <body>
        <h1>Item overview</h1>
        <ul>
          <xsl:for-each select="/Item">
            <li>
              <xsl:value-of select="@name"/>
              :<xsl:value-of select="."/>
            </li>
          </xsl:for-each>
        </ul>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```
To run an XSLT stylesheet over some XML from within XQuery, you need to use an extension module. Extension modules are, well, extensions to the basic XQuery capabilities. eXist has lots of them, and we devote all of Chapter 7 to the subject. An overview (and all function documentation) is accessible through the XQuery Function Documentation app, available through the dashboard.

Transforming documents with XSLT is done with the transform extension module. A little XQuery script that performs this transformation is shown in Example 2-5, and its result in Figure 2-5.

Example 2-5. Using XSLT with the transform extension module

```xquery
xquery version "3.0";

declare option exist:serialize "method=html media-type=text/html";

transform:transform(
    doc("/db/apps/exist-book/getting-started/xml-example.xml"),
    doc("/db/apps/exist-book/getting-started/convert-items.xslt"),
    ()
)
```

Figure 2-5. Result of the XSLT transformation

Notice that the `transform` extension module was not explicitly declared in the XQuery script. eXist does this implicitly for you. The third parameter of `transform:transform`, which here is passed an empty sequence, can contain parameters for the stylesheet.

More about using XSLT transformations within eXist can be found in “XSLT” on page 238.
Hello XInclude

eXist can also do XInclude processing for you. This means that on the way out, when the final results of an XQuery operation are serialized, they are inspected for xi:include elements. When found, these references are expanded.

An interesting feature of the XInclude processing is that you can also refer to XQuery scripts. The script is executed and the result included. Example 2-6 demonstrates this.

Example 2-6. XInclude

```xml
<?xml version="1.0" encoding="UTF-8"?>
<XIncludeEnvelope xmlns:xi="http://www.w3.org/2001/XInclude">
  <xi:include href="xinclude-content.xml"/>
  <xi:include href="hello-world-1.xq"/>
</XIncludeEnvelope>
```

`hello-world-1.xq` is the XQuery script presented in Example 2-3. The included XML file contains the fragment shown in Example 2-7.

Example 2-7. XML fragment to include with XInclude

```xml
<XIncludeContent>This element was included by the XInclude processing in eXist. Yes!</XIncludeContent>
```

Now if you retrieve `xinclude-envelope.xml` from the database, the XInclude references are resolved, resulting in Example 2-8.

Example 2-8. The result of the XInclude processing

```xml
<XIncludeEnvelope xmlns:xi="http://www.w3.org/2001/XInclude">
  <XIncludeContent>This element was included by the XInclude processing in eXist. Yes!</XIncludeContent>
  <results timestamp="2013-02-21T13:12:21.399+01:00">
    <message>Hello XQuery</message>
  </results>
</XIncludeEnvelope>
```

There are more features to XInclude processing, like fallback instructions and the ability to pass parameters to XInclude-d XQuery scripts. Read more about this in “XInclude” on page 243.

Hello XForms

XForms is a W3C standard that defines declaratively the contents of a form on a web page, its behavior, and its result. It’s neither a thick nor a complicated standard.
However, trying to fully understand what’s going on, and all the details (like forms submission), can be challenging!

eXist has two third-party XForms processors built in that you may choose between: betterFORM and XSLTForms. They allow you to create pages that contain XForms logic and have them rendered and executed as the XForms specification describes. To see this in action, take a look at Example 2-9, which will be rendered using betterForm.

Example 2-9. A simple XForms example

```xml
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml"
     xmlns:ev="http://www.w3.org/2001/xml-events"
     xmlns:xf="http://www.w3.org/2002/xforms">
  <head>
    <title>Hello XForms</title>
    <!-- The XForms data model: -->
    <xf:model id="xforms-data-model">
      <xf:submission action="hello-xforms-submit.xq" id="submit-id"
                     method="post"/>
      <xf:instance xmlns=""/>
      <Data>
        <Name/>
        <Date/>
      </Data>
    </xf:instance>
    <xf:bind id="NameBind" nodeset="/Data/Name" required="true()"
            type="xs:string"/>
    <xf:bind id="DateBind" nodeset="/Data/Date" required="true()"
            type="xs:date"/>
  </xf:model>
  </head>
  <!-- -->
  <body>
    <h1>Hello XForms</h1>
    <xf:group>
      <xf:input bind="NameBind">
        <xf:label>Name</xf:label>
      </xf:input>
      <xf:input bind="DateBind">
        <xf:label>Date</xf:label>
      </xf:input>
      <xf:submit submission="submit-id">
        <xf:label>Submit</xf:label>
      </xf:submit>
    </xf:group>
  </body>
</html>
```
This example will let you fill in a simple form. Notice that because we bound the /Data/Date field to the data type xs:date, the form will automatically show a date picker for the date input field! Have a look at the underlying HTML code. As you can see, betterForm adds lots of functionality to make all this happen.

When you press the submit button (after filling in the values), the posted XML will show through the hello-xforms-submit.xq page, as Example 2-10 demonstrates.

Example 2-10. Getting the results of an XForm

```xquery
xquery version "1.0" encoding "UTF-8";

<XFormsResult>
  {request:get-data()}
</XFormsResult>
```

You can find more on XForms in “XForms” on page 254.
This chapter contains an introduction to actually using eXist to build software. It will take you by the hand and guide you step by step along the winding XPath roads, through the XQuery meadows, along some RESTful paths, over the index mountain, and to many, many other wonderful places—all in a quest for the golden ring of an understanding that binds it all (which will definitely not be thrown into some volcano by Gollum).

This chapter guides you through building a simple web application that uses, analyzes, and enables searching of some Shakespeare plays. We start at the very beginning (creating the application’s collections, adding data), and end with a simple but usable piece of software. Along the way, we will tell you where to find more information about the subjects covered. Everything is done natively through eXist; no external IDE is required.

To make ourselves clear: the goal of this chapter is not to teach you XQuery. We assume you have at least a basic understanding of this language, including some experience with XPath. However, we do try to explain what we’re doing with some of the XQuery, so if you don’t know any XQuery yet, do not despair!

**Preparations and Basic Application Setup**

The application we’re going to build needs a set of data to work with. eXist comes with an example application (one of the eXist-db demo apps, available through the dashboard) that contains a nice, consistent, and large enough set of example data to be useful for us. So we’re going to reuse this and, in doing so, show you how to export and import files.

The tools provided alongside eXist for working with the database’s content (see “The Dashboard” on page 24) have the ability to copy content from one database collection
to another. For instance, if you look at the database through the dashboard’s collection browser, you’ll find copy and paste facilities. However, sometimes you have to export and import content from the filesystem, so that’s a good place to start our quest.

**eXist Terminology**

To prevent you from getting confused, let’s introduce some eXist terms to you first:

**Collections**

What you may typically think of as a directory or folder in the filesystem world, is actually called a collection inside eXist’s database. There are some subtle and important differences with collections that you will learn about shortly.

**Resources**

What is called a file in the filesystem world is called a resource inside eXist’s database. Resources can be anything you usually store in a file: images, CSS files, XQuery scripts, and, of course, XML documents.

**Documents**

A resource containing well-formed XML is called a document.

See also “Terminology” on page 88.

**Exporting Documents from eXist**

First, we’re going to export the documents from eXist (afterward, we’ll pretend they came from somewhere else). If you have a WebDAV connection set up (see “Getting Files into and out of the Database” on page 30), this is really easy: just drag and drop the necessary files from the eXist collection to somewhere on your filesystem. If you don’t (yet) have WebDAV working, you can use the Java Admin Client as follows:

1. Start the Java Admin Client by clicking the eXist application icon in your system tray (see “Starting and Stopping eXist with a GUI” on page 23) and choose *Open Java Admin Client*. You’ll see the screen shown in Figure 3-1.
2. Log in as admin (using the password you set when you installed eXist, as described in “Downloading and Installing eXist” on page 19) with the URL `xmldb:exist://localhost:8080/exist/xmlrpc`.

3. After a successful login, the screen shown in Figure 2-4 appears. Navigate to the collection `apps/demo/data`.

4. Select `hamlet.xml` and click the menu command File→Export a resource to a file…. Dump it on your disk somewhere. Repeat this for `macbeth.xml` and `r_and_j.xml` (guess what: Romeo and Juliet!).

5. Change the filename of the Romeo and Juliet play from `r_and_j.xml` to `r and j.xml` (replace the underscores with spaces). This will illustrate an important property of eXist later on.

So now we have three Shakespeare plays in XML markup on our disk. Let’s pretend these files came from somewhere else and import them into our database.
Designing an Application’s Collection Structure and Importing Data

When you write an application in eXist (or anywhere else), you need a place to store your code and the accompanying data. Now, in eXist you can design any collection structure you like, but it is customary to store applications underneath the /db/apps collection. So, that is where we’re going to put our 101 application.

In this example, we’ll consider our data, the plays, static (immutable) data. However, most applications also have dynamic data (data your application creates, updates, uploads, etc.). There is a debate as to whether (a subcollection of) /db/apps/<yourapp> is a good location for this data. Some application designers argue that you should be consistent and keep everything in one place. But storing your dynamic data somewhere else (e.g., in /db/data/<yourapp>) has benefits of its own. For instance, you can more easily update your application’s code without losing the accumulated dynamic data. We won’t worry about this in our 101 course, but make sure to give it some thought if you’re going to build a real application.

You could use the Java Admin Client again to create the necessary collections and import the plays (there’s an Import Files button in the toolbar), but let’s check out another useful tool. Close the Java Admin Client, open a web browser, and follow these steps:

1. Browse to the dashboard (http://localhost:8080/exist/).
2. Log in as admin (click on “Not logged in” in the upper-left corner).
3. Click on the Collections tile.
4. Navigate to /db/apps.
5. Create a fresh /db/apps/exist101 collection (the New Collection command is in the toolbar).
6. Navigate into the /db/apps/exist101 collection and create the /db/apps/exist101/data collection.
7. Navigate to the /db/apps/exist101/data collection and click “Upload resources” in the toolbar.
8. Upload the plays we just downloaded into the collection.

The collection browser should now look like Figure 3-2.
Look at the name of the *Romeo and Juliet* file. Instead of *r and j.xml*, it is now called *r%20and%20j.xml*. What happened? Well, names of collections and resources inside eXist are always *URL-encoded URIs*. Reserved characters, according to the URL encoding rules, are percent-encoded. A space character is one of these, so that explains the %20 codes. More about this can be found in “Use URIs” on page 91. We’ll come back to how to handle these names later.

**Viewing the Data**

Let’s pretend for a moment we did not have the data on our disk before we imported it into the database. Instead of viewing a file on disk, how can we view XML (and other) resources stored in eXist?

The easiest and most versatile way of working with stored data is through an editor that is connected with eXist. Luckily for us, eXist has a built-in IDE, *eXide*, which we can use to view and edit files as follows:

2. If you’re not already logged in, log in as admin (click on “Not logged in” in the upper-left corner).
3. Click on the “eXide - XQuery IDE” tile. eXide will open in a new browser window or tab.
4. Click Open, navigate to /db/apps/exist101/data, and open *hamlet.xml* (Figure 3-3).
Alternatively, you can view the XML file in a browser through eXist’s REST interface (see “Querying the Database Using REST” on page 94). Simply visit http://localhost:8080/exist/rest/db/apps/exist101/data/hamlet.xml.

But wait, it’s not working! You should get an error message about a shakes.xsl stylesheet not being found. Crime doesn’t pay: it’s our punishment for being a data thief!

The problem is that the XML files we started with were part of an application and were coupled to an XSL stylesheet by a processing instruction. This stylesheet, meant to create an HTML version of the play, was not copied by us and therefore, alas, could not be found. There’s more about using XSLT via processing instructions in “Invoking XSLT by Processing Instruction” on page 242.

Fear not; the problem is easily solved. Open hamlet.xml (again) in eXide and look at the first line. It begins with:

```xml
<?xml-stylesheet href="shakes.xsl" type="text/xsl"?>
```

Remove this processing instruction using eXide, save the file, and try the URL again. You should now see hamlet.xml in all its XML glory.
If you feel bold enough, you could also try this: do not remove the processing instruction (or use one of the other files) and copy the `shakes.xsl` file from `/db/apps/demo/data` to `/db/apps/exist101/data`, like we did with the plays. When you now open the document in your browser (with the URL given before), you'll see a nicely rendered HTML page.

**Listing the Plays (XML)**

Let's write our first XQuery script and have it find out which plays we have in the `/db/apps/exist101/data` collection. For now we'll return the result as an XML fragment, and in the next section we'll create a nice-looking HTML page from this. There is more than one way to do this (where have I heard that line before?), and we'll show you a few.

First, perform the following preparations:

2. If you're not already logged in, log in as `admin` (click on “Not logged in” in the upper-left corner).
3. Click on the “eXide - XQuery IDE” tile. eXide will open in a new browser window or tab.
5. Click Save and save the script as `/db/apps/exist101/playlist.xq`.

**Listing with the collection Function**

In our first version of the solution, we will use the XPath `collection` function to iterate over all the resources in our `data` collection. Enter the following code and press Run:

```xml
for $resource in collection("/db/apps/exist101/data")
return base-uri($resource)
```

The `collection` function returns a sequence of documents (document-node items) for all the resources found in the given collection and its subcollections (for details, see “The collection Function” on page 93). The XPath `base-uri` function returns the URI for a document-node, which in eXist is the path leading to the resource.

Running this should return (in the bottom window) a list with the plays, including each one’s full path in the database.

Turning this list of strings into a well-formed XML fragment is simple:
<plays>
{
  for $resource in collection("/db/apps/exist101/data")
  return <play url="(base-uri($resource))"/>
}
</plays>

This should return:

<plays>
  <play uri="/db/apps/exist101/data/hamlet.xml"/>
  <play uri="/db/apps/exist101/data/macbeth.xml"/>
  <play uri="/db/apps/exist101/data/r%20and%20j.xml"/>
</plays>

Now let’s tweak this a little more. Later on we’re going to present a list of available plays to the user, and it would be nice if we could display the name of the document (without a collection path) and, more importantly, the name of the play.

To get the name of the file, we have to do some string munging on the URI we already have. For this, XPath’s regular expressions come in handy. Regular expressions are expressions to parse and work with strings. We won’t explain them here because that would take too much ink (in fact, there are whole books devoted to them). For now, just accept that the following expression returns the part of a filename after the last / character:

replace(base-uri($resource), '.+/(.+)$', '$1')

There is one more thing about the name: we don’t want the URI encodings, like %20, to show up in them. For this, we use a native eXist function from one of its extension modules (extension modules are covered in Chapter 7): util:unescape-uri. It needs two parameters: the name to unescape and the character encoding, which is nowadays almost always UTF-8. With this, the full code to get a nicely formatted filename becomes:

util:unescape-uri(
  replace(base-uri($resource), '.+/(.+)$', '$1'),
  'UTF-8'
)

To get the name of the play, we have to dive into the XML of the play itself. It’s always in the /PLAY/TITLE element, and since we already have the root document-node, retrieving this is a piece of cake:

$resource/PLAY/TITLE/text()

Using the /text function will ensure we get the result as a text node. If we didn’t use this, the TITLE element itself would be returned (try it out if you want to).

With all this, our full code becomes what you see in Example 3-1.
Example 3-1. The full code to get the play information

```xquery
xquery version "3.0";
<plays>
  { for $resource in collection("/db/apps/exist101/data")
    return <play uri="{base-uri($resource)}"
      name="{util:unescape-uri(replace(base-uri($resource), ".+/(.+)$", "$1"), "UTF-8")}">
    { $resource/PLAY/TITLE/text() }
  </play>
} </plays>
```

And, with some slight improvements (using variables to store repeating data), it looks like Example 3-2.

Example 3-2. Improved version of the code that gets the play information

```xquery
xquery version "3.0";
<plays>
  { let $data-collection := "/db/apps/exist101/data"
    for $resource in collection($data-collection)
    let $uri := base-uri($resource)
    return <play uri="{$uri}" 
      name="{util:unescape-uri(replace($uri, ".+/(.+)$", "$1"), "UTF-8")}">
    { $resource/PLAY/TITLE/text() }
  </play>
} </plays>
```

Don’t forget to save your code. After that, you can also try it from the browser. Try the URL `http://localhost:8080/exist/rest/db/apps/exist101/playlist.xq`. You should see the following:

```xml
<plays>
  <play uri="/db/apps/exist101/data/hanlet.xml" name="hanlet.xml">
    The Tragedy of Hamlet, Prince of Denmark
  </play>
  <play uri="/db/apps/exist101/data/macbeth.xml" name="macbeth.xml">
    The Tragedy of Macbeth
  </play>
  <play uri="/db/apps/exist101/data/r%20and%20j.xml" name="r and j.xml">
    The Tragedy of Romeo and Juliet
  </play>
</plays>
```
Listing with the xmldb Extension Module

Another way to return a list is by using one of eXist’s extension modules, xmldb (see “Controlling the Database from Code” on page 107). An extension module contains functions that perform actions that are difficult or impossible to do in standard XQuery. eXist has quite a lot of them, and Appendix A lists the most important ones. If you want to explore the wonderful features that extension modules have to offer, you can access their documentation through the dashboard’s XQuery Function Documentation tile.

Instead of the code entered in the previous section, try the following:

```
xmldb:get-child-resources("/db/apps/exist101/data")
```

Your result window should now show a list of the play’s resource names (but without their paths). So, where the collection function returned a document-node sequence, xmldb:get-child-resources returns a string sequence. To get from a string to inside the XML (to get the name of the play), we have to resolve a document-node from this string. For this we use the XPath doc function (see “The doc Function” on page 94).

So, without further ado, Example 3-3 shows a piece of code that returns exactly the same results as the code in Example 3-2, but by a different means.

Example 3-3. Getting the play information using an extension function

```
xquery version "3.0";
<plays>
{ 
  let $data-collection := "/db/apps/exist101/data"
  for $resource-name in xmldb:get-child-resources($data-collection)
  let $uri := concat($data-collection, '/', $resource-name)
  return 
    <play uri="{$uri}"
      name="{util:unescape-uri($resource-name, "UTF-8")]"> 
    { 
      doc($url)/PLAY/TITLE/text()
    }
  </play>
}
</plays>
```

Listing the Plays (HTML)

As a next step, let’s present this information to the user as a nicely formatted HTML page. For this we have to (you might have guessed) create the HTML ourselves, including pieces like headers. For now we won’t bother to make it look fancy by using CSS and the like, but of course you can add that too if you want.
First, let’s set up a basic HTML (or more precisely, XHTML) page without much content to see how this works. Enter the code in Example 3-4 and save it as plays-home.xq.

Example 3-4. Basic HTML page code

```xml
xquery version "3.0";
<html>
<head>
  <meta HTTP-EQUIV="Content-Type" content="text/html; charset=UTF-8"/>
  <title>Our Shakespeare plays</title>
</head>
<body>
<h1>Our Shakespeare plays</h1>
</body>
</html>
```

Now run it. In the bottom part of the eXide screen, you’ll see the same unexciting piece of XHTML as you entered. Running it from the browser (http://localhost:8080/exist/rest/db/apps/exist101/plays-home.xq) doesn’t make it look any better. So what’s missing?

What’s missing here is that you have to tell eXist that this is XHTML and that it should serialize it as such. There is more than one way to do this (see “Serialization” on page 118), but the easiest is to add an exist:serialize option in the XQuery prolog to tell eXist to serialize the query result as XHTML and send it to the browser as an HTML page and not as a bare piece of XML:

```
declare option exist:serialize "method=xhtml media-type=text/html";
```

While we’re changing things anyway, let’s also get rid of the double entry for the page’s title and put this in a global variable. Example 3-5 shows our improved basic HTML page.

Example 3-5. Improved version of the basic HTML page code

```xml
xquery version "3.0";

declare option exist:serialize "method=xhtml media-type=text/html";
declare variable $page-title := "Our Shakespeare plays";

<html>
<head>
  <meta HTTP-EQUIV="Content-Type" content="text/html; charset=UTF-8"/>
  <title>{$page-title}</title>
</head>
<body>
  <h1>{$page-title}</h1>
</body>
</html>
```
When you run this from within eXide nothing too exciting happens, but when you try it from the browser, you see the result shown in Figure 3-4.

![Our Shakespeare plays](localhost:8080/exist/rest/db/apps/exist101/plays-home.xql)

**Figure 3-4. The exciting output of our first basic HTML page**

OK, now we’re getting somewhere! Now let’s use the output from “Listing the Plays (XML)” on page 45 to display a list of plays (we’ll use the code with the collection function here, but the code that uses the `xmldb:get-child-resources` function is also fine). With a little copy and pasting, the code in Example 3-6 is constructed quickly.

*Example 3-6. Code for the HTML page that returns a list of plays (plays-home.xq)*

```xquery
xquery version "3.0";

declare option exist:serialize "method=xhtml media-type=text/html";

declare variable $page-title := "Our Shakespeare plays";

let $play-info :=
  <plays>
    { for $resource in collection('/db/apps/exist101/data')
      return
        <play uri="{base-uri($resource)}"
          name="{util:unescape-uri(replace(base-uri($resource), ".+/(.+)$", "$1"), "UTF-8")}"
          >
          { $resource/PLAY/TITLE/text() }
        </play>
      }
    </plays>

return
  <html>
    <head>
      <meta HTTP-EQUIV="Content-Type" content="text/html; charset=UTF-8"/>
      <title>{$page-title}</title>
    </head>
    <body>
  </body>
```

50 | Chapter 3: Using eXist 101
And in the browser, you should see the output in Figure 3-5.

![Our Shakespeare plays](image)

**Figure 3-5. Our HTML page with a list of plays**

Those new to XQuery might wonder why on earth our HTML code (the part that starts with `<html>`) is now suddenly underneath a `return` statement. This is because we introduced a local variable (`let $play-info := …`). As soon as you do this, your code becomes an XQuery FLWOR expression and needs a `return` statement for the part you want to return.

### Analyzing the Plays

Now assume we’re a famous play director and in need of some data about the play. Among the many, many questions directors struggle with are “Which character has the most lines?” and “How many actors do I need?” Let’s find out the answers using XQuery.

Assume we have the document-node for the play’s XML to analyze in a variable called `$play-document`. Subproblems to solve for our analysis are:

- We need a list of characters who speak. If you look inside the play’s XML, you’ll see that everything said is inside a `SPEECH` element with a `SPEAKER` subelement. So, to get a list of different speakers, all you have to do is get all the different values of `SPEECH/SPEAKER` and filter them for uniqueness:

  ```xquery
  distinct-values($play-document//SPEECH/SPEAKER)
  ```
• To show totals and compute percentages for each speaker, we need the full set of spoken lines in the play:

\[
\text{let } \$\text{all-lines} := \$\text{play-document}//\text{SPEECH/LINE}
\]

• We need the lines spoken by each speaker. Assuming the name of the speaker is in \$speaker, we can get these with:

\[
\$\text{speaker-lines} := \$\text{play-document}//\text{SPEECH[SPEAKER eq } \$\text{speaker}]/\text{LINE}
\]

• Given a sequence of LINE elements in a variable \$elms (the sequence of LINE elements we retrieved in one of the previous two bullets), how many words are spoken? A rough but usable approximation for this can be calculated by tokenizing everything said using whitespace boundaries and counting/aggregating the results:

\[
\text{sum}(\$\text{elms} ! \text{count}(\text{tokenize(., '\text{s+}')}))
\]

This might need a little further explanation:

— The exclamation mark after the \$elms expression is an XQuery 3.0 “bang” or “simple map” operator (see “The simple map operator” on page 114). It performs the operation on the right for all members of the sequence on the left.

You could have done this in several other (and probably more customary) ways (e.g., using a FLWOR expression), but this seemed a useful way to introduce one of the new XQuery 3.0 capabilities.

— The \text{tokenize} function tokenizes (breaks up) strings on boundaries given by a regular expression. The regular expression '\text{s+}' signifies a sequence of whitespace characters, so that gives us the words (more or less, sometimes punctuation gets in the way, but let’s forget about that for now).

— We’re not interested in the words themselves but only in how many there are. So we simply count them.

— The outer \text{sum} function sums the numeric results of what’s inside, returning the total of all words spoken in the elements in \$elms.

As a last step, let’s put this functionality into a local function (because we’re going to use it twice in our code: once for the full play and once for every speaker):

\[
\text{declare function } \text{local:word-count($elms as element()*) as xs:integer}
\{
  \text{sum($elms ! count(tokenize(., '\text{s+}')}))}
\};
\]

Now let’s put this all together and create a page that shows us the results of our analysis (just for \textit{Hamlet}) in a table. The code for this is shown in Example 3-7.
Example 3-7. Code to analyze Hamlet

xquery version "3.0";

declare option exist:serialize "method=xhtml media-type=text/html";
declare variable $page-title as xs:string := "Play analysis";
declare variable $play-uri as xs:string := "/db/apps/exist101/data/hamlet.xml";

declare function local:word-count($elms as element()*) as xs:integer
{
    sum($elms ! count(tokenize(., "\W+")))
};

let $play-document := doc($play-uri)
let $play-title := string($play-document//PLAY/TITLE)
let $speakers := distinct-values($play-document//SPEECH/SPEAKER)
let $all-lines := $play-document//SPEECH/LINE
let $all-word-count := local:word-count($all-lines)
return
<html>
<head>
    <meta HTTP-EQUIV="Content-Type" content="text/html; charset=UTF-8"/>
    <title>{$page-title}</title>
</head>
<body>
    <h1>{$page-title}: {$play-title}</h1>
    <p>Total lines: {count($all-lines)}</p>
    <p>Total words: {$all-word-count}</p>
    <p>Total speakers: {count($speakers)}</p>
    <br/>
    <table border="1">
        <tr>
            <th>Speaker</th>
            <th>Lines</th>
            <th>Words</th>
            <th>Perc</th>
        </tr>
        { for $speaker in $speakers
            let $speaker-lines :=
            $play-document//SPEECH[SPEAKER eq $speaker]/LINE
            let $speaker-word-count := local:word-count($speaker-lines)
            let $speaker-word-perc :=
            ($speaker-word-count div $all-word-count) * 100
            order by $speaker
            return
            <tr>
                <td>{$speaker}</td>
                <td>{count($speaker-lines)}</td>
                <td>{$speaker-word-count}</td>
                <td>{format-number($speaker-word-perc, "0.00")]%}</td>
            </tr>
        }
    </table>
</body>
</html>
Save this code as `analyze-play.xq`. In the browser (http://localhost:8080/exist/rest/db/apps/exist101/analyze-play.xq), the result looks like Figure 3-6.

**Play analysis: The Tragedy of Hamlet, Prince of Denmark**

Total lines: 4014  
Total words: 29549  
Total speakers: 35

<table>
<thead>
<tr>
<th>Speaker</th>
<th>Lines</th>
<th>Words</th>
<th>Perc</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>4</td>
<td>11</td>
<td>0.04%</td>
</tr>
<tr>
<td>BERNARDO</td>
<td>38</td>
<td>220</td>
<td>0.74%</td>
</tr>
<tr>
<td>CORNELIUS</td>
<td>1</td>
<td>10</td>
<td>0.03%</td>
</tr>
<tr>
<td>Captain</td>
<td>12</td>
<td>84</td>
<td>0.28%</td>
</tr>
<tr>
<td>Daines</td>
<td>3</td>
<td>13</td>
<td>0.04%</td>
</tr>
<tr>
<td>FRANCISCO</td>
<td>10</td>
<td>54</td>
<td>0.18%</td>
</tr>
<tr>
<td>First Ambassador</td>
<td>6</td>
<td>40</td>
<td>0.14%</td>
</tr>
<tr>
<td>First Clown</td>
<td>93</td>
<td>733</td>
<td>2.48%</td>
</tr>
<tr>
<td>First Player</td>
<td>52</td>
<td>365</td>
<td>1.21%</td>
</tr>
<tr>
<td>First Priest</td>
<td>13</td>
<td>90</td>
<td>0.30%</td>
</tr>
<tr>
<td>First Sailor</td>
<td>5</td>
<td>39</td>
<td>0.13%</td>
</tr>
<tr>
<td>GUILDENSTERN</td>
<td>53</td>
<td>339</td>
<td>1.15%</td>
</tr>
<tr>
<td>Gentleman</td>
<td>24</td>
<td>176</td>
<td>0.60%</td>
</tr>
<tr>
<td>Ghost</td>
<td>95</td>
<td>677</td>
<td>2.29%</td>
</tr>
<tr>
<td>HAMLET</td>
<td>1495</td>
<td>11542</td>
<td>39.06%</td>
</tr>
<tr>
<td>HORATIO</td>
<td>291</td>
<td>2049</td>
<td>6.93%</td>
</tr>
</tbody>
</table>

**Figure 3-6. Partial output of the analysis for Hamlet**

Well, that wasn’t too hard, was it? The analysis itself was almost laughably easy; most of the code is actually dedicated to creating a nicely formatted page.

As you might have noticed, we used a different approach here than in **Example 3-6**. There, we first created an XML fragment and later used it to create the HTML. In the analysis page, we put the FLWOR expressions directly inside the HTML. This was done intentionally, to show you two different approaches.
By the way: maybe you shouldn’t go for the role of Hamlet—almost 1,500 lines with 11,500 words is a lot to learn by heart. And have mercy on the director: finding and directing 35 actors is not exactly a walk in the park!

**Linking the Analysis to the Play Overview**

At this moment our application still consists of two different pages: an overview and an analysis page for *Hamlet*. So how can we tie these together and use the analysis page for every play?

To do this, we first need to link from our overview to our analysis page and pass the URI of the play to analyze in the link. Go back to our basic home page (Example 3-6) and change the line starting with `<li>` as follows:

```
<li>{string($play)} {{string($play/@name)}}
  <a href="analyze-play.xq?uri={encode-for-uri($play/@uri)}">analysis</a>
</li>
```

The `a` element (that produces an HTML link) links to the analysis page and passes the URI of the play to analyze in the link. Go back to our basic home page (Example 3-6) and change the line starting with `<li>` as follows:

```
<li>{string($play)} {{string($play/@name)}}
  <a href="analyze-play.xq?uri={encode-for-uri($play/@uri)}">analysis</a>
</li>
```

The `a` element (that produces an HTML link) links to the analysis page and passes the URI of the play to analyze in the link. The `encode-for-uri` function is necessary here because the URI passed contains characters that are misinterpreted if we pass them straight: `encode-for-uri` creates `%` encodings for them, e.g., `%2F` for a slash (`/`) character.

An interesting phenomenon occurs when you use this technique for *Romeo and Juliet*. Remember, the URI for this play already contains URI-escaped characters (the spaces in the resource name: `r%20and%20j.xml`). Passing this URI through the `encode-for-uri` function means the URI will become double URI encoded! The resource name is passed as `r%2520and%2520j.xml`.

The second thing we have to do is change our analysis page to retrieve the URI in the `uri` parameter and act on this. For this, change the declaration of the `$play-uri` variable to:

```
declare variable $play-uri as xs:string := request:get-parameter('uri', ());
```

`request:get-parameter` is an extension function from the request extension module (see “The request Extension Module” on page 209) that returns the value of a parameter passed in the URL (or of a control in an HTML form–driven request). The second parameter, which in this example is the empty sequence `()`, can be used to pass a default value.

After making these changes you can analyze not only *Hamlet*, but all three of the plays (and assuming you can provide the data, then many more). Neat, isn’t it?
Searching the Plays

The next two enhancements add search functionality to our little play application. One uses straight XQuery (actually as an example of how not to do it), and the other uses eXist’s full-text indexing capabilities.

Searching Using Straight XQuery

Let’s assume we need to search the plays for certain words or phrases. A first naive approach could be to use just straight XQuery. The searching is not hard at all. For instance, try the following surprisingly short code snippet, which searches all the plays for the word fantasy (and adds a play attribute to tell us which play the results came from):

```
for $line in collection('/db/apps/exist101/data')//SPEECH/LINE[contains(., 'fantasy')]
return <LINE play="{base-uri($line)}">{string($line)}</LINE>
```

This shows that Hamlet is three times as fantastic as Romeo and Juliet, and there’s no fantasy in Macbeth (for the Shakespearians: that was a joke):

```
<LINE play="/db/apps/exist101/data/hamlet.xml">
Horatio says 'tis but our fantasy,</LINE>
<LINE play="/db/apps/exist101/data/hamlet.xml">
Is not this something more than fantasy?</LINE>
<LINE play="/db/apps/exist101/data/hamlet.xml">
That, for a fantasy and trick of fame,</LINE>
<LINE play="/db/apps/exist101/data/r%20and%20j.xml">
Begot of nothing but vain fantasy,</LINE>
```

Although the searching itself is simple to program, we do need some code to tie it all together. First, add a search form to the main page. Reopen `plays-home.xq` (Example 3-6) and add the following lines between the `<ul>` and `</body>` closing elements:

```
<br/>
<h3>Search using XQuery:</h3>
<form method="POST" action="search-1.xq">
  <input type="text" name="searchphrase" size="40"/>
  <input type="submit" value="Search!"/>
</form>
```

When you reopen the page in your browser (http://localhost:8080/exist/rest/db/apps/exist101/plays-home.xq) you’ll see a nice little search form with a Search! button.

Now we have to create an XQuery script that does something with the search request and displays the results. Open a new XQuery file in eXide, save it as `search-1.xq`, and enter the code shown in Example 3-8.
Example 3-8. Search page that uses straight XQuery

```xquery
xquery version "3.0";

declare option exist:serialize "method=xhtml media-type=text/html";
declare variable $page-title := "Search results with XQuery";
declare variable $searchphrase := request:get-parameter("searchphrase", (

<html>
  <head>
    <meta HTTP-EQUIV="Content-Type" content="text/html; charset=UTF-8"/>
    <title>{$page-title}</title>
  </head>
  <body>
    <h1>{$page-title}</h1>
    <p>Search phrase: "{$searchphrase}"</p>
    <ul>
      {for $line in collection("/db/apps/exist101/data")//SPEECH/LINE
       [contains(., $searchphrase)]
        return
        <li>
          from: {string(root($line)/PLAY/TITLE)}<br/>
          <i>{string($line)}</i>
        </li>
      }
    </ul>
  </body>
</html>
```

When you now enter `fantasy` as a search phrase and press Search!, the result shown in Figure 3-7 should appear.

![Search results with XQuery](localhost:8080/exist/rest/db/apps/exist101/search-1.xql)

**Figure 3-7. Results of searching the plays for “fantasy”**
Well, that’s not a bad result for a fairly minimal amount of code. But, we can do better:

- This solution doesn’t scale well. Searching like this works by comparing the query string with each word in the text: all `<LINE>` elements are string-searched for the search phrase. When you get more and more data, the searches will become slower and slower.
- It searches for literal strings, which means it’s case-sensitive and will also return results where the search phrase is part of a word (e.g., searching for `faun` will also return lines with `fauna`).
- It cannot handle search expressions (e.g., searching for lines with `fantasy` and `Macbeth`).

So, let’s add a second search facility with some enhancements.

**Searching Using an Index**

An index in a database is comparable to the index in a book: it allows you to quickly find something based on an index key. Although useful and often even necessary, using indexes is not without disadvantages: it slows down creating and updating data and consumes (a little bit of) disk space. However, indexes make searching much faster, especially for large collections of documents.

eXist supports several types of indexes, which we will elaborate on in Chapters Chapter 11 and Chapter 12. In this example we’re going to use a full-text index.

To illustrate things, let’s first add all the code before creating the index. Again, we have to change `plays_home.xq` and add a new search form. Reopen `plays-home.xq` (Example 3-6) and add the following lines right before the closing `<body>` element:

```xml
<h3>Search using index:</h3>
<form method="POST" action="search-2.xq">
  <input type="text" name="searchphrase" size="40"/>
  <input type="submit" value="Search!"/>
</form>
```

Now create `search-2.xq`, as shown in Example 3-9. It’s almost the same as `search-1.xq` (Example 3-8), so copy, paste, and fiddle is probably a good option here.

**Example 3-9. Search page that uses indexed search**

```xml
xquery version "3.0";

declare option exist:serialize "method=xhtml media-type=text/html";
declare variable $page-title := "Search results with XQuery using full-text index";
declare variable $searchphrase := request:get-parameter("searchphrase", ());
```
Its main difference from `search-1.xq` is the way the search expression is formulated:

```
collection('/db/apps/exist101/data')//SPEECH/LINE[ft:query(., $searchphrase)]
```

The `ft:query` function looks for an index on the elements in its first argument. However, since this index is not yet there, the result set will currently still be empty (which we can easily prove by searching on `fantasy` using the indexed search). So, let’s define this index.

Index definitions (and some other content) are kept in what is best described as a “shadow” database collection structure underneath `/db/system/config`. If you go there (using, for instance, the collection browser of the dashboard), you’ll see parts of the main database collection structure there too, starting with `/db`.

To add our play index definition, create the collection `/db/system/config/db/apps/exist101/data`. In this collection create an XML document called `collection.xconf` with the contents shown in Example 3-10.

Example 3-10. The index definition `collection.xconf` document

```
<collection xmlns="http://exist-db.org/collection-config/1.0">
  <index xmlns:tei="http://www.tei-c.org/ns/1.0">
    <lucene>
      <text qname="LINE"/>
    </lucene>
  </index>
</collection>
```
We’re almost done. After defining a new collection, you’ll have to reindex. Use the collection browser to go to /db/apps/exist101/data and click “Reindex collection” (the second button from the left). This will take a few seconds. After this initial reindex you’ll never have to do any manual maintenance again; eXist now knows the index and will keep it up to date for you when the dataset changes.

Our indexed search is ready to go. Try it by searching on fantasy again, and you should now see the same results as in our straight XQuery search (Figure 3-7).

So, have we gained anything? Yes, we have!

- Search on FANTASY: the search is now case-insensitive.
- Search on fantasy horatio. You’ll get a long list with lines in which one or both of the two words are present.
- Search on fantasy AND horatio. Now there is only one line: the only one with both words.
- But perhaps most importantly: this search is faster (although you might not notice on modern hardware with this small dataset), and it scales! Add more plays, and it will stay fast.

This was the just tip of the iceberg in terms of what you can do with indexes. For example, in “Full-Text Index and KWIC Example” on page 285, you’ll learn about sorting by relevance and how to use eXist’s “keywords in context” feature to highlight the matching words in the search results. Adding such capabilities to our eXist 101 application is not difficult once you have progressed further through the book, and we’ll leave this to you as an independent exercise.

Creating a Log

The last capability we’re going to add to our application is having it maintain a logfile, showing you how you can update the database from XQuery. Since we’ll probably want to use this feature from more than one XQuery script, we’re going to put the code for it into an XQuery module and show you a little about modularization. While testing, we’ll also run into some interesting security problems and show you how to solve them.

By way of preparation, create the collection /db/apps/exist101/log to hold our log information.

We require our logging to write strings to a logfile and add a timestamp. If the logfile does not exist, it should be created. The logfile should be easy to use and callable from all our existing XQuery scripts without code duplication.
This functionality calls for a design that uses an XQuery module. The module should contain an XQuery function that performs the logging. To make it easy to use, this function must return zilch (a.k.a. the empty sequence in XPath lingo), so we can simply call it from anywhere without influencing our output. Example 3-11 is a module that does exactly this. Create this module and save it as /db/apps/exist101/log-module.xqm.

Example 3-11. The logging module log-module.xqm

```xquery
xquery version "3.0" encoding "UTF-8";

module namespace x101log = "http://www.exist-db.org/book/namespaces/exist101";

declare function x101log:add-log-message($message as xs:string)
  as empty-sequence()
{
  let $logfile-collection := "/db/apps/exist101/log"
  let $logfile-name := "exist101-log.xml"
  let $logfile-full := concat($logfile-collection, '/', $logfile-name)
  let $logfile-created :=
    if(doc-available($logfile-full))
      then $logfile-full
    else
      xmldb:store($logfile-collection, $logfile-name, <eXist101-Log/>)
  return
    update insert
    <LogEntry timestamp="{current-dateTime()}">{$message}</LogEntry>
  into doc($logfile-full)/*
};
```

1. The module namespace definition at the top defines the code as an XQuery module. A module must have a namespace, and it's customary to use something that starts with a URL you own (to avoid name clashes). What comes after that is up to you. Don't let the starting http:// fool you: it's just a string without any further special meaning.

2. We declare a function that returns empty-sequence(). In other words: nothing.

3-5. The first three let statements in the function simply define the logfile's location, name, and full name.

6. let $logfile-created := ... checks whether or not the logfile exists. If not, it uses the xmldb:store extension function to create a new logfile with an empty <eXist101-Log/> root element. If the logfile already exists, it simply emulates the return value of xmldb:store by returning the logfile's name.
Have a look at the rest of the code. We never actually use this $logfile-created variable, so if eXist did lazy evaluation (computing values only when needed), the logfile would never get created. Lucky for us, eXist always evaluates variable assignments from top to bottom, which we exploit here to allow side effects. Read more about this in “XQuery Execution” on page 118.

The return part of the function’s FLWOR expression contains an eXist update insert statement that inserts a LogEntry element, with a timestamp and the given text, as a child of the root element of our logfile. An update statement always returns an empty sequence, so that takes care of our required empty return value. More about eXist XQuery update functionality can be found in “Updating Documents” on page 101.

Let’s call this logging function from our home page. To do this, first add the following import module statement at the top of plays-home.xq (see Example 3-6), directly after the xquery version "3.0"; declaration:

```xquery
import module namespace x101log="http://www.exist-db.org/book/namespaces/exist101" at "log-module.xqm";
```

Now plays-home.xq “knows” about the logging module. To call the logging function, add the following line directly before the let $play-info := part (don’t forget the finishing comma!):

```xquery
x101log:add-log-message('Visited plays-home'),
```

Let’s test our changes. You might be logged in as admin, but to make a point we would like you to do this without being logged in. Please log out and close your browser, reopen it, and run plays-home.xq by visiting http://localhost:8080/rest/db/apps/exist101/plays-home.xq. Oops—you get an error stating, “Write permission is not granted on the Collection.” Why is that?

eXist has a strict security system. When you’re logged in as admin you can do anything, but when you’re not it’s a different game. All access to collections and resources must abide by the strict eXist security rules, which look, not at all by accident, remarkably like those in a Unix environment.

To view the relevant security settings, open the collection browser in the dashboard and navigate to /db/apps/exist101. The permissions on your log collection (db/apps/exist101/log) should read crwxr-xr-x (that is, if you haven’t changed any defaults). You’ll find a complete explanation of what this means in Chapter 8. For now, we’ll focus on what’s important for us.

When you visit eXist without being logged in, you’re using a built-in account called guest. This is, as we can see, not the owner or the group the collection belongs to, so
the relevant security settings are the last three characters of the permissions: r-x. This means anybody can list and open the collection, but not write to it. Let’s change this so we can at least write to the collection and create our logfile.

Use the collection browser or the Java Admin Client to set the permissions to crwxr-xrwx—that is, set the Write permission for Other. Figure 3-8 shows how this looks in the collection browser.

![Figure 3-8. Changing the permissions of the log collection](image)

After that, log out and close your browser (to stop being admin), open it again, and revisit the plays-home.xq page. This should now run smoothly. Look inside the log collection, and there it is: our exist101-log.xml document. It contains something like:

```xml
<eXist101-Log>
  <LogEntry timestamp="2013-04-09T20:51:27.205+02:00">Visited plays-home</LogEntry>
</eXist101-Log>
```

And yes, every time you revisit the plays-home.xq page, a new LogEntry element is added. I’ll leave it up to you to add the logging code to the other pages of our small but beautiful application.

The x permission on a collection means something different than the x permission on a resource. For a resource it means execute rights, and this is important for XQuery scripts. Try this out by, for instance, revoking the x permissions on our plays-home.xq file. You should now get a security warning when you try to run it.
What’s Next?

Although we could go on (and on and on), this is the end of our 101 course. We hope it gave you at least a taste of what’s possible, and some pointers on how to start out with eXist. There is still much more to learn (that’s why there are chapters after this one)! Here are some suggestions for further exploration:

- Explore eXist’s extension modules’ functionality using the information in Chapter 7 and through the dashboard’s XQuery Function Documentation browser.
- Tighten security to a specific set of users with the information in Chapter 8.
- Change our little demo application into a real one using RESTXQ or URL rewriting. This will give you much more control over the URLs needed to visit the pages, security, error handling, and so on. More information can be found in Chapter 9.
- Explore other index types and settings, as described in Chapters Chapter 11 and Chapter 12.
- Use one of the other supported XML technologies from Chapter 10—for instance, creating a PDF version of our analysis report with XSL-FO.
- Integrate the application with the rest of the world using the technologies described in Chapter 13.

But most of all, remember to have fun!
Most of your work using eXist will be done in the XQuery programming language. This chapter covers what is and is not supported. It will also describe some eXist XQuery specifics, like controlling serialization and available pragmas.

**eXist’s XQuery Implementation**

Currently, 2.0+ versions of eXist support almost the full XQuery 1.0 specification (as eXist has done for years) and quite a lot of XQuery 3.0. This section will provide you with the details.

**XQuery 1.0 Support**

eXist implements almost all of the full XQuery 1.0 specification, with the following exceptions:

- eXist’s XQuery processor does not support the schema import and schema validation features. This is perfectly reasonable as they are defined as optional in the XQuery specification (`validate` and `import schema`). The database does not store type information along with the (values of) nodes; consequently it cannot know the typed value of a node and has to assume `xs:untypedAtomic`. This is compliant with the behavior defined by the XQuery specification.

- You cannot specify a data type in an element or attribute test. eXist supports the node test `element(test-node)`, but the test `element(test-node, xs:integer)` results in a syntax error.
The absence of the features does not mean that eXist is not type-safe; it is, very much so. It only means that type checking based on schema imports is not implemented.

eXist tested its implementation against the official XQuery Test Suite (XQTS version 1.0.2). Of the more than 14,000 tests, it passed over 99%.

eXist does not yet type-check the name of an element or attribute. So, strangely enough, you can write let $elm as element(a) := <b/> and eXist will find it absolutely OK, although this is a relaxation from the XQuery specification. The advice is not to use name tests in element or attribute data type specifications, though. So, use element() or attribute() instead of element(a) or attribute(b), since specifying a name implies type checking that alas never occurs.

**XQuery 3.0 Support**

New since version 2.0 is eXist’s support for XQuery 3.0. As of writing, this specification had reached Proposed Recommendation status and several partial implementations were available.

To enable the XQuery 3.0 support, start your XQuery program with:

```xml
xquery version "3.0";
```

XQuery 3.0 is a relatively new and probably not yet very well known standard. Therefore, the support eXist offers is handled in somewhat more detail next. For the exact details, please refer to the standard.

**XPath 3.0 functions**

Many of the extra functions defined in XPath and XQuery 3.0 are implemented. Among them are some very useful ones, like `format-dateTime`.

An exact list of what is available and what isn’t can be found with the XQuery Function Documentation browser in the dashboard. Browse the [http://www.w3.org/2005/xpath-functions](http://www.w3.org/2005/xpath-functions) module.

**try/catch**

The XQuery 3.0 try/catch mechanism allows you to catch errors raised during execution. These can be errors raised by the XQuery engine (meaning your code did something wrong), or errors you’ve explicitly raised with the `error` function. The
following example shows a try/catch usage example that sets a variable to -1 if a 
division-by-zero error occurs:

\[
\text{let } \$\text{result as xs:decimal} := \\
\text{try} \\
\{ \\
\text{\$something div \$something-else} \\
\} \\
\text{catch } \text{err:FOAR0001} \{ -1 \}
\]

This example tests for a very specific error, which is good, because we would like a 
warning when something unexpectedly goes wrong. If you want to test for \textit{all} errors 
that can occur, change the \texttt{err:FOAR0001} into an \texttt{*}.

If you want to test for a specific error condition but don't know its 

code, probably the easiest way to find it is to force the error and 
copy the error code reported back into the XQuery.

Inside the catch operand you have access to information about the error through a 
number of implicitly declared variables—\$err:code, \$err:line-number, 
\$err:column-number, \$err:description, \$err:value, \$err:module, and \$err:additional. Please refer to the XQuery 3.0 specification for full details.

\textbf{switch expression}

The XQuery 3.0 \texttt{switch} expression implements that which in other languages is often 
called a case expression or, in XSLT, an \texttt{xsl:choose}. This example was copied from 
the XQuery 3.0 specification:

\[
\text{switch (}\$\text{animal}) \\
\text{case } "\text{Cow}" \text{ return } "\text{Moo}" \\
\text{case } "\text{Cat}" \text{ return } "\text{Meow}" \\
\text{case } "\text{Duck}" \text{ return } "\text{Quack}" \\
\text{default return } "\text{What's that odd noise?}" 
\]

\textbf{Higher-order functions}

A higher-order function is a function that takes another function as a parameter or 
returns a function. The normal use case for this is mapping or filter functions. Here is 
an example:

\[
\text{declare function local:map-example($func, $list) } \{ 1 \} \\
\text{for $item in $list} \\
\text{return} \\
\text{$func($item)} \\
\};
\]
let $f := \text{upper-case}\#1$
return
local:map-example($f, ("Hello", "world!"))

1. We first define a function, local:map, that runs the function passed in its first parameter, $\text{func}$, over all members of its second operand, $\text{list}$.

2. We then assign the upper-case function to $f$. The #1 after the function name means that we want the upper-case function with only one parameter (in case there are more).

3. Finally, we call local:map with our function and some input strings. It returns the expected HELLO WORLD!.

Higher-order functions is a serious subject in its own right and includes topics such as: inline and partial functions, closures, currying, and more. For further information, you can refer to the XQuery 3.0 specification and to this excellent eXist wiki article.

The simple map operator

The XQuery 3.0 bang operator ! (or simple map operator, as it is officially called) can be seen as a shorthand for simple FLWOR expressions. It applies the right-hand expression to each item in the sequence gained from evaluating the left-hand expression. For instance:

$\text{(1 to 10) ! . + 1}$

is the same as:

for $i$ in (1 to 10) return $i + 1$

The string concatenation operator

The string concatenation operator || is a shorthand replacement for the concat function: it concatenates strings. For example, the following expression will be true:

'Hello ' || 'world' eq concat('Hello ', 'world')

Annotations

XQuery 3.0 allows annotations for functions and variables. This is used, for instance, to make them private (visible only in the enclosing module) or public:

declare %private variable $myns:only-i-can-see-this := 'secret';

declare %public function myns:do-something-public() {
Within eXist, annotations are also used for RESTXQ (see “Building Applications with RESTXQ” on page 215).

Serialization

eXist now supports the new XQuery 3.0 manner of controlling serialization. For instance, this:

```xml
declare namespace output = "http://www.w3.org/2010/xslt-xquery-serialization";
declare option output:method "xml";
declare option output:media-type "text/xml";
```

is exactly the same as (eXist’s incumbent nonstandard mechanism):

```xml
declare option exist:serialize "method=xml media-type=text/xml";
```

The options supported are the same also. More about serialization and the full list of options supported can be found in “Controlling Serialization” on page 119.

The group by clause

eXist has had an order by clause for its FLWOR expressions since 2006. Unfortunately, this was not compatible with the XQuery 3.0 group by clause, and so it was replaced in the 2.0 release with the official version. Here is an example:

```xml
let $data as element()* := (  
  <item>Apples</item>,  
  <item>Bananas</item>,  
  <item>Apricots</item>,  
  <item>Pears</item>,  
  <item>Brambles</item>  
)
return <GroupedItems>  
{  
  for $item in $data  
  group by $key := upper-case(substring($item, 1, 1))  
  order by $key  
  return <Group key="{$key}">
  {$item}
  </Group>
}
</GroupedItems>
```

The fruits are grouped and sorted based upon the uppercased first characters of their names. This returns:
<GroupedItems>
  <Group key="A">
    <item>Apples</item>
    <item>Apricots</item>
  </Group>
  <Group key="B">
    <item>Bananas</item>
    <item>Brambles</item>
  </Group>
  <Group key="P">
    <item>Pears</item>
  </Group>
</GroupedItems>

Other XQuery Extras

Beside eXist’s support for XQuery 1.0 and the majority of XQuery 3.0, it also has a few interesting features which are currently specific to its XQuery implementation.

The map data type proposed for XQuery 3.1

The `map` data type in eXist is essentially a key/value lookup table. Keys must be atomic values (e.g., `xs:string`, `xs:integer`, `xs:date`, etc.). Values can be anything from a simple numbers to complete XML documents. Here is a basic example of creating and using a map:

```
let $map1 := map {
"a" := 1,
"b" := <XML>this is <i>cool</i></XML>
} return ($map1("a"), $map1("b"))
```

This will return:

```
1
<XML>this is <i>cool</i></XML>
```

Working programmatically with maps is possible through the `map` extension module. This module allows you to do everything from checking for the existence of keys up to constructing maps on the fly. Please refer to the online function documentation for more information.

Maps are immutable, like any other XQuery variables. So, changing a map using the functions from the `map` extension module (e.g., calling `map:remove`) will create a new map.

There is also an article about maps on the eXist wiki.
Java binding

eXist allows you to make arbitrary calls to Java libraries using the so-called Java binding. For example:

```xquery
declare namespace javasystem="java:java.lang.System";
declare namespace javamath="java:java.lang.Math";

javasystem:getenv('JAVA_HOME'),
javamath:sqrt(2)
```

For security reasons, the Java binding is disabled by default. If you want to use it, edit $EXIST_HOME/conf.xml, search for the enable-java-binding attribute, set its value to "yes", and restart eXist for the change to take effect.

There are some specifics you need to know about when using the Java binding:

- If the function name in XQuery contains a hyphen, the hyphen is removed and the character following it is converted to uppercase. So, a call in XQuery to `toString` will call the Java method `toString`.
- Java constructors can be called using the `new` function.
- eXist adds a generic type, `object`, to its data-model, which is used for all Java objects.
- Instance methods of a class (methods that work on a specific object, like most of the Java methods) must get the object reference as their first parameter.
- When a method returns an array, it is converted to a sequence and you can iterate over it using a FLWOR expression.

Here is an example that will return a list containing the names of all files and subdirectories in the $EXIST_HOME directory:

```xquery
declare namespace javafile="java:java.io.File";

let $fobject as object := javafile:new(system:get-exist-home())
return
for $file in javafile:list($fobject)
return
$file
```

If you only want to get a list of files and directories, it is probably easier to use the `file` extension module instead of the Java binding.
**XQuery Execution**

There are some details you should be aware of regarding XQuery execution in eXist. These include:

*Transaction boundaries*

eXist is transactional only during updates to the database; that is, a single update either succeeds or fails atomically, not something in between, even if a crash occurs in the middle of the operation.

eXist is *not* transactional during the execution of a full XQuery script (like some other XQuery engines are). An XQuery script does not run in isolation, and updates made by it or by concurrently running neighbor scripts will immediately be visible. However, you can group updates into a single transaction; see the `exist:batch-transaction` pragma in “eXist XQuery Pragmas” on page 123.

*Evaluation of expressions*

eXist does *not* lazily evaluate expressions. For instance, a series of `let` expressions will all be evaluated, from top to bottom, even if some of the variables are never used again.

The reasoning behind this has to do with side effects, which XQuery officially doesn’t have, but which (as we all know) in a real-world program are a necessity. For instance, when you have a function that adds to a logfile, you want it executed even if you don’t do anything with its return value.

As a consequence, be careful computing expensive values that might never be used. It’s better to either defer this until you really need them (e.g., nesting them inside an `if-then-else` structure) or do something along the lines of:

```xquery
let $expensive-value :=
  if (...decide-whether-value-is-really-needed...) then compute-value...
  else ()
```

**Serialization**

Although it may seem as though eXist works directly with XML, as in “text with a lot of angle brackets,” it does not. Internally, XML is represented as an efficient tree-structured data type. Only on the way out, in the final step, are the angle brackets added and the XML displayed once again as we know it. This process of changing the internal representation into something suitable for the outside world is known as serialization.

Controlling serialization is important: sometimes you may want XML, while at other times you want HTML and/or JSON. You may perhaps also want to set the Internet media type explicitly, or control indentation.
For the XSLT programmers among us who think this sounds familiar: you’re right. In XSLT, serialization is controlled likewise through the `xsl:output` element.

**Controlling Serialization**

There are a number of ways you can control serialization from within your XQuery scripts:

```xquery
option exist:serialize
```

You can control serialization by adding a `declare option exist:serialize` statement to the XQuery prolog. For instance:

```xquery
declare option exist:serialize
  "method=html media-type=text/html indent=no";
```

The contents of the `exist:serialize` option are a whitespace-separated list of name/value pairs, as described in the following section. You do not have to define the `exist` namespace prefix. eXist automatically binds this to the appropriate `http://exist.sourceforge.net/NS/exist` namespace.

```xquery
util:get-option, util:declare-option
```

These extension functions allow you to inspect and set the value of an XQuery script option programmatically. For instance, setting the serialization options can be done with:

```xquery
util:declare-option("exist:serialize",
  "method=html media-type=text/html indent=no")
```

**XQuery 3.0 serialization settings**

eXist now also supports the standard XQuery 3.0 way of controlling serialization. This is described in “Serialization” on page 115.

**Serialization Options**

This section will list all the serialization options that eXist supports.

**General serialization options**

The more general serialization options closely mimic the options of the same name available on the XSLT `xsl:output` command:

```xquery
method=xml|microxml|xhtml|html5|text|json
```

Sets the principal serialization method.

The `microxml` method produces MicroXML as opposed to full XML. You can find out more about MicroXML from the [W3C MicroXML Community Group](https://www.w3.org/Community/MicroXML/).
The xhtml method makes sure that only the short form is used for elements that are declared empty in the XHTML specification. For instance, a br element is always returned as \(<br/>\). In addition, if you omit the XHTML namespace from your XML, you can have the XHTML serializer inject it for you by setting the serialization option enforce-xhtml=yes.

If you specify the text method, only the atomized content of elements is returned: for example, \(<\text{foo}>this is content</\text{foo}>\) will return \(this \text{ is content}\). Namespaces, attributes, processing instructions, and comments are ignored.

For JSON and JSONP serialization options, see “JSON serialization” on page 121.

- **media-type=...**
  Indicates the Internet media type of the output. This is used to set the HTTP Content-Type header if the query is running in an HTTP context.

- **encoding=...**
  Specifies the character encoding used for serialization. The default is the encoding set in the XQuery declaration at the top of the program. If that is not set, the default is UTF-8.

- **indent=yes|no**
  Indicates whether the output should be indented.

- **omit-xml-declaration=yes|no**
  Specifies whether the XML declaration (\(<?xml version="1.0"?>\)) at the top of the output should be omitted.

- **doctype-public=... doctype-system=...**
  When at least one of these is present, a doctype declaration is included at the top of the output.

- **enforce-xhtml=yes|no**
  Forces all output to be in the XHTML (http://www.w3.org/1999/xhtml) namespace.

### Post-processing serialization options

eXist can do post-processing of the XQuery result by processing xi:include elements and \(<\text{xml-stylesheet?}>\) processing instructions referencing XSLT stylesheets. You can control this with the following options:

- **expand-xincludes=yes|no**
  Indicates whether the serializer should process any xi:include elements (see “XInclude” on page 243). The default is yes.
process-xsl-pi=yes|no
Indicates whether the serializer should process any `<?xml-stylesheet
  type="text/xsl" href="..."?>` processing instructions (see “Invoking XSLT
  by Processing Instruction” on page 242). The default is yes.

eXist-specific serialization options

eXist-specific options include the following:

add-exist-id=element|all
  If you output elements that come from the database, eXist will add an attribute
  exist:id to them, showing the internal node identifier of each element. Setting
  this option to element will show only the node identifier of the top-level element;
  setting it to all will show all node identifiers.

  There are functions in the util extension module to work with these identifiers.

highlight-matches=both|elements|attributes|none
  When querying text with the full-text or NGram extensions, the query engine
  tracks the exact position of all matches inside text content. The serializer can
  later use this information to mark those matches by wrapping them into an
  exist:match element. Find more information about this in “Locating Matches”
  on page 296.

JSON serialization

JSON (JavaScript Object Notation) is a lightweight data-interchange format. eXist
has a JSON serializer built in that you can enable by setting the serialization method
to json (see “General serialization options” on page 119). There is one related seriali-
zation option:

jsonp=...
  Produces JSONP (JSON with padding) output by wrapping the JSON output in
  the named function. For example, specifying jsonp=abc causes the output to be
  wrapped in the JavaScript function abc like so: abc({"hello": "world"}). This
  can be useful when you’re working around same-origin policies in some web
  browsers.

  It is also possible to set the JSONP function dynamically by calling the function
  util:declare-option and passing in the function name; for example,
  util:declare-option("exist:serialize", "method=json jsonp=myFunction
  Name").

Here is a summary of how eXist performs the JSON serialization (see also the wiki
entry on this subject):
- The root element is absorbed: `<root>A</root>` becomes "A".
- Attributes are serialized as properties, with the attribute name and its value.
- An element with a single text child becomes a property whose value is the text child: `<e>text</e>` becomes `{"e": "text"}`.
- Sibling elements with the same name within a parent element are added to an array: `<A><b>1</b><b>2</b></A>` becomes `{ "b" : ["1", "2"] }`.
- In mixed-content nodes, text nodes are dropped.
- If an element has attribute and text content, the text content becomes a property: `<A a="b">1</A>` becomes `{ "A" : { "a" : "b", "#text" : "1" } }`.
- An empty element becomes `null`: `<e/>` becomes `{"e": null}`.
- An element with name `<json:value>` is serialized as a simple value, not an object: `<json:value>my-value</json:value>` becomes "my-value".

Sometimes it is necessary to ensure that a certain property is serialized as an array, even if there’s only one corresponding element in the XML input, you can use the attribute `json:array="true|false"` for this.

By default, all values are strings. If you want to output a literal value—for example, to serialize a number—use the attribute `json:literal="true"`.

The JSON prefix `json` should be bound to the namespace `http://www.json.org`. As an example, here is some XML:

```xml
<Root xmlns:json="http://www.json.org">
  <Items>
    <Item id="1">Bananas</Item>
    <Item>CPU motherboards</Item>
  </Items>
  <Items>
    <Item json:array="yes">Bricks</Item>
  </Items>
  <Mixed>This is <i>mixed</i> content</Mixed>
  <Empty/>
  <Literal json:literal="yes">1</Literal>
</Root>
```

And here is its JSON serialization:

```json
{ "Items" : [  { "Item" : [  { "id" : "1", "#text" : "Bananas" },
    "CPU motherboards"] },
  {"Item" : ["Bricks"] }],
  "Mixed" : { "i" : "mixed" },
  "Empty" : null,
  "Literal" : 1 }
```

In addition to the JSON serializer in eXist, which attempts to convert XML into JSON with as little effort from the developer as possible, there are three other XQuery modules that enable you to work with JSON. The first two modules—JSON XQuery
(see json) and JSONP XQuery (see jsonp)—work in much the same way as the JSON serializer. The third module, XQJSON (see xqjson), which was written by John Snelson and adapted by Joe Wichtenowski, is the newest JSON addition to eXist; it allows you to serialize XML to JSON as well as parse JSON back into XML so that you can round-trip your data.

## Controlling XQuery Execution

There are several mechanisms which give you control over the execution of your XQuery scripts.

### eXist XQuery Pragmas

With XQuery pragmas, you can set implementation-specific options for parts of your code. The general syntax is:

```xml
(# pragmaname #) {
    (: Your XQuery code block :) 
}
```

eXist has the following pragmas:

**exist:batch-transaction**

Provides a way to combine multiple updates on the database into a single transaction. Only works for updates done through eXist’s XQuery update extension. For example:

```xml
(# exist:batch-transaction #) {
    update delete $document/*/LogEntry[position() ge 10],
    update insert $new-entry preceding $document/*/LogEntry[1]
}
```

**exist:force-index-use**

Useful for debugging index usage (see Chapters Chapter 11 and Chapter 12). Will raise an error if there is no index available for the given XQuery expression. This can help you to check whether indexes are correctly defined.

**exist:no-index**

Prevents the use of indexes on the given XQuery expression. Useful for debugging or for curiosity purposes (“How long does my query take without indexes?”). Also, sometimes it is more efficient to run without indexes than with—for instance, when a search isn’t very selective.

**exist:optimize**

Enables optimization for the given XQuery expression. If you’ve turned optimization off (with declare option exist:optimize "enable=no";), as discussed
in “Serialization Options” on page 119), you can turn it on again for specific expressions with this pragma.

exist:timer
Measures the time it takes to execute the XQuery expressions within the pragma—for instance, (# exist:timer #) { count(//TEST) }.

To see the timer, you need to enable tracing in the $EXIST_HOME/log4j.xml configuration file (set <priority value="trace"/> for the root logger). You’ll see an entry like this in the $EXIST_HOME/webapp/WEB-INF/logs/exist.log file:

2012-09-12 15:01:29,846 [eXistThread-31] TRACE (TimerPragma.java [after]:63)
- Elapsed: 171ms. for expression: count([root-node]/descendant::{}TEST)

Limiting Execution Time and Output Size

You can control execution time and query output size by adding the correct declare option exist:... statement to the XQuery prolog:

declare option exist:timeout "time-in-msecs"
Indicates the maximum amount of time (specified in milliseconds) that a query is allowed to execute for. If this is exceeded, an error will be raised.

declare option exist:output-size-limit "size-hint"
Defines a limit on the maximum size of created document fragments. This limit is an estimation, specified in terms of the accumulated number of nodes contained in all generated fragments. If this is exceeded, an error will be raised.

Other Options

Here are some miscellaneous options you can set by adding a declare option exist:... statement to the XQuery prolog:

declare option exist:implicit-timezone "duration"
Specifies the implicit time zone for the XQuery context as defined in the XQuery standard. More information is available at http://www.w3.org/TR/xquery/#dt-timezone.

declare option exist:current-dateTime "dateTime"
If for some reason you don’t want to use your operating system’s date/time, you can specify your own using this option (it is merely there to enable some of the XQuery test suite cases to run).

declare option exist:optimize "enable=yes|no"
Use this to disable the query optimizer in eXist (the default is yes, of course). This is linked to the exist:optimize pragma; see “eXist XQuery Pragmas” on page 123.
XQuery Documentation with xqDoc

xqDoc is an effort to standardize XQuery documentation in a similar vein to how JavaDoc has for Java. xqDoc works by reading specialized comments you insert into your XQuery code. A parser can then use these to extract additional information about your module, its (global) variables, and its functions. This information could then, for example, be used to display details about a module to the user. The eXist function browser is a good example of an implementation which uses xqDoc to achieve exactly that.

Here is an example of a little module containing xqDoc information:

```xml
<xquery version "1.0" encoding "UTF-8">

(:~
  :  Example module with xqDoc information
  ;
  :  @version 1.0
  :  @author Erik Siegel
  ;)

module namespace xquerydoc="http://www.exist-db.org/book/XQueryDoc";

(:~
  :  Example dummy function
  ;
  :  @param $in The input to the function
  ;)

declare function xquerydoc:test($in as xs:string +) as xs:string
{
  'Dummy'
};
```

All comments starting with (~ are parsed by the xqDoc parser. Keywords in these comments start with an @ character. The exact syntax can be found on the xqDoc website.

eXist has an inspect extension module to work with xqDoc. The functions in this module return an XML representation of the module’s content, including possible annotations by the xqDoc comments. For instance, running inspect:inspect on the preceding example module returns:

```
<module uri="http://www.exist-db.org/book/XQueryDoc" prefix="xquerydoc">
  <description> Example module with xqDoc information </description>
  <author> Erik Siegel </author>
  <version> 1.0 </version>
  <variable name="xquerydoc:global" type="xs:string" cardinality="exactly one"/>
  <function name="xquerydoc:test"
    module="http://www.exist-db.org/book/XQueryDoc">
    <argument type="xs:string" cardinality="one or more" var="in">
      The input to the function</argument>
    <returns type="xs:string" cardinality="exactly one"/>
```
<description> Example dummy function </description>
</function>
</module>

From this XML you could easily create any required HTML or PDF documentation.
eXist does not support the full xqDoc specification. If you need some specific xqDoc feature, please run some tests to see if it is present.
Text Encoding Fundamentals: Element list

Elements for basic TEI documents

This is more of a brief referencesheet than an exhaustive list of TEI elements: it is intended to provide you with a way to look up the most commonly used elements, grouped together for the exercises in which we'll be encountering them. For detailed information about the contents and semantics of these elements (and for other more arcane elements), have a look at the TEI Guidelines [http://www.tei-c.org/release/doc/tei-p5-doc/en/html/].

Element groups

structure
  <TEI>, <back>, <body>, <front>, <group>, <teiHeader>, and <text>

general purpose block-level
  <ab>, <argument>, <div>, <head>, <item>, <label>, <list>, <p>, <quote>, and <said>

general purpose phrase-level
  <bibl>, <date>, <distinct>, <emph>, <foreign>, <hi>, <mentioned>, <name>, <q>, <quote>, <rs>, <said>, <seg>, <soCalled>, and <term>

poetry
  <l>, <lg>, and <rhyme>

drama
  <castGroup>, <castItem>, <castList>, <role>, <roleDesc>, <sp>, <speaker>, and <stage>

diary entries, letters, etc.
  <closer>, <dateline>, <opener>, <postscript>, <salute>, <signed>, and <trailer>

alternative transcriptions
  <abbr>, <choice>, <corr>, <expan>, <orig>, <reg>, and <sic>

manuscripts and physicality of documents
  <add>, <addSpan>, <cb>, <del>, <delSpan>, <handShift>, <lb>, <milestone>, and <pb>

editorial annotation
  <app>, <damage>, <gap>, <lem>, <rdg>, <restore>, <subst>, <supplied>, and <unclear>

hypertextual
  <anchor>, <note>, <ptr>, and <ref>
Elements (in alphabetical order)

<TEI>
The outermost (or ‘root’) element for any TEI P5 conformant document. It groups together the TEI header and the document text. It must have the TEI namespace specified, and should have an xml:lang= attribute, i.e. <TEI xmlns="http://www.tei-c.org/ns/1.0" xml:lang="en">.

<ab>
An ‘anonymous block’, that is, a paragraph-like chunk that does not carry the semantic weight of a paragraph. Use type= and maybe subtype= to categorize.

<abbr>
An abbreviation; may be used alone or, when inside <choice>, in combination with <expan> which holds an expanded reading.

<add>
A handwritten addition. The hand= attribute indicates the handwriting in which the addition is made. This attribute contains an identifier which points to a <hand> element in the <profileDesc> of the TEI header; this <hand> element contains an extended description of the handwriting, ink, and other details.

<addSpan>
An empty element which marks the starting point for a handwritten addition that either is too long to be encoded with <add>, or overlaps an element boundary. Its spanTo= attribute points to an <anchor> element which marks the endpoint of the added material. The hand= attribute indicates the handwriting in which the addition is made (see above for details).

<anchor>
An anchor point, usually used as a place for some other element (such as a note) to point to, using the anchor’s xml:id= attribute.

<app>
Contains one entry in a critical apparatus, with an optional lemma and at least one reading.

<argument>
A short summary or description of the contents of the following section. Contains one or more <p> or <lg> elements.

<back>
Contains the back matter of the document, if any: indices, appendices, epilogues, colophons, errata lists, etc. May be subdivided into <div> elements if necessary.

<bibl>
Used to encode bibliographical references, either in a list (using <listBibl>) or in running prose.

<body>
Contains the main body of the document, not including front matter and back matter. The <body> element typically includes one or more <div> elements. It may start with a <head>. (Think about where the <head> belongs—is it the heading for the body, or the heading for the first division?)

<castGroup>
A grouping of related items in a cast list, containing one or more <castItem> elements and an optional <head> and <trailer>.
<castItem>
  An item in a cast list, containing a <role> and an optional <roleDesc>.
</castItem>

<castList>
  A cast list in a dramatic text, listing the roles in the drama. It consists of one or more
  <castItem> or <castGroup> elements.
</castList>

<cb>
  An empty element which marks the break between one column and the next. Equivalent to
  <milestone unit="column">.
</cb>

<choice>
  Groups together two or more alternate encodings of a phrase-level passage, using the elements
  listed below.
</choice>

<closer>
  Very similar to <opener>, but located at the end of the <div> instead of at the beginning.
</closer>

<corr>
  A corrected reading of a typographical error or oddity in the original; may be used alone or,
  when inside <choice>, in combination with <sic>, which holds the original reading.
</corr>

<damage>
  A damaged portion of the original text; the type= attribute allows you to classify the damage,
  and the extent= attribute allows you to indicate the extent of the damage.
</damage>

<date>
  Used to encode dates. The when= attribute can be used to encode a regularized form of the
  date (e.g. <date when="2001">The first year of the new century</date> or
  <date when="2005-05-29">Sun, 29 May 05</date>).
</date>

<dateline>
  Used within <opener> and <closer> to encode the date and place of writing. Contains words
  and phrase-level encoding.
</dateline>

<del>
  A deletion. The hand= attribute indicates the handwriting in which the addition is made (see
  above for details).
</del>

<delSpan>
  An empty element which marks the starting point for a deletion that is either too long to be
  encoded with <del> or that overlaps an element boundary. Its spanTo= attribute points to an
  <anchor> element which marks the endpoint of the deleted material. The hand= attribute
  indicates the handwriting in which the deletion is made (see above for details).
</delSpan>

<distinct>
  Used for linguistically distinct words (e.g. dialect words, regionally accented words).
</distinct>

<div>
  A division of a text: for instance, an act, a chapter, a section, a poem, a letter
  Use the type= attribute to indicate what kind of division.
</div>

<emph>
  Used to encode emphasized words or phrases.
</emph>

<expan>
  The expanded reading of an abbreviation; typically used inside <choice>, in combination with
  <abbr> which holds the corresponding abbreviated reading. Rarely used alone.
</expan>
<foreign>
Used for foreign-language words when no other element (e.g. <quote> ) is already present.
</foreign>

<front>
Contains the front matter of the document, if any: title pages, tables of contents, introductory essays, and so forth. The <front> element contains an optional <titlePage> and may be subdivided into <div> elements.
</front>

<gap>
A gap in the original text (either from damage, deletion, excerption, or some other cause). The <desc> child element provides a description of what is missing, and the reason= attribute provides the reason for the omission.
</gap>

<group>
This element is used to represent documents which contain more than one independent text. It appears instead of <body> in the overall TEI document structure, and groups together multiple <text> elements, with an optional <front> and <back>.
</group>

<handShift>
An empty element which marks the boundary point at which a change of handwriting takes place. Its new= attribute indicates the handwriting that begins at the point being marked. The new= attribute functions just like the hand= attribute, in pointing to a <hand> element in the TEI header, which provides detailed information on the handwriting in question.
</handShift>

<head>
The heading of a division: contains words and phrase-level encoding. <head> may appear at the start of <div>, but also at the start of <body>, <front>, <back>, <list>, and <lg>.
</head>

<hi>
Used to encode words or phrases which are highlighted for reasons which the encoder either does not know or chooses not to analyse.
</hi>

<item>
An item in a list: contains an optional <label> followed by words and phrase-level encoding, or a series of paragraphs.
</item>

<l>
A single verse line: contains words and phrase-level elements. May have a met= attribute to formally specify the metrical pattern.
</l>

<label>
The label of an item (e.g. a letter, number, or word indicating its order or other facts about it): contains words and phrase-level encoding. Note that <label> can also be the first element inside a paragraph.
</label>

<lb>
An empty element which marks a typographical line break. Equivalent to <milestone unit=”line”>.
</lb>

<lem>
A lemma; e.g., the reading from the base text.
</lem>

<lg>
A group of verse lines: contains one or more <l> elements. May have a rhyme= attribute to formally specify the rhyme scheme, e.g.<lg rhyme=”ABAB”>.
</lg>

<list>
A list: contains a series of <item> elements.
<mentioned>
Used for words which are mentioned but not used (for instance, for spelling or definition purposes).

<milestone>
An empty element which marks a boundary point in the text according to some standard reference system, such as signatures, scrolls, leaves. Use the unit= attribute to indicate the reference system whose units are being marked at this point.

<name>
Used to encode all kinds of names, i.e. proper nouns and noun-phrases. If you want to distinguish between different kinds of names, you can use the type= attribute (e.g. <name type="person">). TEI also includes specific elements for different kinds of names (e.g. <persName>) for projects that need more detailed encoding. The <rs> element is a more generic version of <name>, which may be used to encode common nouns and noun phrases.

<note>
A note (a footnote, endnote, marginal note, or inline note). Link the note to the point where it’s anchored using xml:id= and target=. <note> contains most anything, including words and phrase-level encoding, or one or more <p> elements.

<opener>
This element may appear at the start of a <div>, <text>, <front>, or <back>, and it groups together the elements that appear at the start of a letter or similar document: the date and place of writing (using <dateLine>, and the salutation to the person being addressed (using <salute>).

<orig>
An unmodernized reading in the original; may be used alone or, when inside <choice>, in combination with <reg>, which holds a regularized reading.

<p>
A prose paragraph: contains words and phrase-level encoding.

<pb>
An empty element which marks the break between one page and another. By convention, information stored in the attributes of <pb> refer to the page that follows the break. Equivalent to <milestone unit="page">.

<ptr>
Indicates a reference to some other XML element (either in the current document or some other accessible document) by pointing to it with a URI on the target= attribute. Must not have content. E.g., <ptr target="#art08_sec08" />.

<postscript>
Used to encode a postscript, e.g. of a letter

<q>
Used to encode passages surrounded by quotation marks, when you don’t want to bother with a more precise element like <said>. Roughly the same as <hi rend="surrounded-with-quotation-marks">.

<quote>
Used to encode quotations from other sources; contains words and phrase-level encoding.

<rdg>
A single reading, e.g. from a particular witness.
<ref>
Indicates a reference to some other XML element (either in the current document or some other accessible document) by pointing to it with a URI on the target= attribute. May (and probably should) have content. E.g., <ref target="#art08_sec08">the <soCalled>IP</soCalled> clause</ref>.
</ref>

<reg>
A modernization of a reading in the original; may be used alone or, when inside <choice>, in combination with <orig>, which holds the corresponding unmodernized reading.
</reg>

<restore>
Indicates restoration of text to an earlier state by cancellation of a marking or instruction; in particular, useful to indicate that a deletion was restored, e.g. by the notation 'stet'.
</restore>

<rhyme>
May be optionally used to indicate the portion of the metrical line that rhymes, and with its label= attribute which part of the rhyme scheme is in play.
</rhyme>

<role>
The name of a role in a cast list.
</role>

<roleDesc>
The description of a role in a cast list.
</roleDesc>

<rs>
Used to encode all kinds of references to people, places, and things; i.e., nouns and noun phrases. If you want to distinguish between different categories of entity being referred to, you can use the type= attribute (e.g. <rs type="person">). The <name> element is a more specialized version of <rs>, reserved for proper nouns and noun-phrases.
</rs>

<said>
Passages spoken aloud or thought, e.g. by a character in a novel.
</said>

<seg>
General-purpose phrase-level segment: use type= and maybe subtype= to categorize.
</seg>

<salute>
Used within <opener> and <closer> to encode the salutation to the person being addressed (e.g. "Dear Sir", or "I remain faithfully yours "). Contains words and phrase-level encoding.
</salute>

<sic>
A typographical error or oddity in the original; may be used alone or, when inside <choice>, in combination with <corr>, which holds a corrected reading.
</sic>

<signed>
Used within <closer> to encode the signature or name of the person writing. Contains words and phrase-level encoding.
</signed>

<soCalled>
Used to encode (or express) authorial distance; e.g., phrases that were or should be in scare quotes.
</soCalled>

<sp>
A dramatic speech; usually begins with a <speaker> element, followed by a <p> or <lg>.
</sp>

<speaker>
A speaker identification printed in the text.
</speaker>

<stage>
A stage direction. The type= attribute may be used to identify the kind of stage direction; suggested values include:
business
costume
delivery
entrance
exit
location
narrative
novelistic

<subst>
Groups together an <add> and a <del> so that the addition is understood as being a substitution for the deletion.

<supplied>
 Indicates that a given word or passage cannot be read in the original and is being supplied (either through editorial judgment or from some other textual source).

<teiHeader>
The wrapper for all of the document’s metadata. The elements that go inside the TEI header are too numerous to list usefully here; see the templates for details.

<term>
Used to encode specialized terminology; often associated with a <gloss>.

<text>
The wrapper element which contains all of the document’s content. The <text> element is most often used for a single work (i.e. a single published document, or a single aesthetic unit such as a play or a work of fiction). Terms like “single work” and “aesthetic unit” need to be defined by the individual project. A <text> element contains an optional <front>, a mandatory <body>, and an optional <back>.

<trailer>
This element is used for things that come at the very end of the document or section, such as “The End”.

<unclear>
Indicates that a given word or passage is unclear, but not entirely illegible (expresses uncertainty rather than absolute lack of information); multiple alternative readings may be grouped in a <choice> element.

Attributes (in alphabetic order)

met=
May be used to specify the metrical pattern a line (or line group).

n=
Provides a label or identifier for this particular element, not necessarily unique.

next= and prev=
Allow what is logically a single text object (e.g. a quotation) to be encoded as a series of two or more discrete XML elements, as a work-around for overlap problems. These attributes represent the connections between these fragmentary elements, by pointing to a prior or subsequent element in the chain of fragments. They do so by referring to that element’s xml:id= value.
That is, if `next=` is specified on a `<said>` element, then its value should be a hash mark (#) followed by the value of the `xml:id=` of another `<said>` element, the one that is the next part of the spoken passage. For example,

```xml
<said xml:id="s01" next="#s02">Hey</said>, he said, <said xml:id="s02" prev="#s01">What's up?</said>.
```

**rend=**
May be used to specify how the element looked in the source. E.g., `<head rend="align (center)"` or `<l rend="indent(2)"`.

**rhyme=**
May be used to specify the rhyme scheme of a linegroup.

**target=**
Provides a URI (e.g. `http://bauman.zapto.org/gallery/Niagara_Falls_2008-01/2008_01_07T16_35_39` or `#sect08`) that points to either another document or an element within an XML document (including the current one).

**xml:id=**
Provides a unique identifier for this particular element, thus allowing other elements to point to it (using their `target=`, `next=`, `prev=`, etc.).

**xml:lang=**
Used to indicate the language of an element's content. Its value conforms to BCP 47 (a standard system for defining language codes). For information on how BCP 47 codes are constructed, see the note in the [data.language] documentation. Some sample values for the `xml:lang=` attribute are:

<table>
<thead>
<tr>
<th>Language</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>English</td>
<td>en</td>
</tr>
<tr>
<td>French</td>
<td>fr</td>
</tr>
<tr>
<td>German</td>
<td>de</td>
</tr>
<tr>
<td>Italian</td>
<td>it</td>
</tr>
<tr>
<td>Latin</td>
<td>la</td>
</tr>
<tr>
<td>Arabic as spoken in Iraq</td>
<td>ar-IQ</td>
</tr>
<tr>
<td>Chinese</td>
<td>zh</td>
</tr>
<tr>
<td>simplified Chinese</td>
<td>zh-Hans</td>
</tr>
<tr>
<td>Taiwanese</td>
<td>zh-TW</td>
</tr>
</tbody>
</table>

If further explanation is required, a `<language>` element with an `ident=` attribute of the same BCP 47 code can be specified in the TEI header.

Some Online References and Resources

Example Sites
The Old Bailey Online
http://www.oldbaileyonline.org/
- An online edition of historical criminal trials, which will be the source for exercises used in class.

Robert Graves Diary
http://graves.uvic.ca/graves/site/index.xml

Thoureau’s Kalendar
http://thoreauscalendar.umf.maine.edu/index.html

Map of Early Modern London
https://mapoflondon.uvic.ca/

XML Ecosystem
Technical Reports of the W3C
https://www.w3.org/
- The W3 Consortium publishes guidelines on XML and related technologies. (These are meant for advanced users, but worth checking out as you learn.)

HTML / CSS / Web Dev.
Introductions and Reference:
http://book.mixu.net/css/
- Great reference for CSS layout.

- More of an intro to the new features that were introduced with CSS3, but potentially useful.

- Beginner’s Guide to CSS3

- A useful CSS Properties Reference. Each element links to W3C, with additional useful info.

- A Codecademy course which combines basic HTML, jQuery, JavaScript. Possibly useful to practice what you learned in this class. (Free, with some advanced features for a charge.)
Mozilla’s developer reference is fantastic.

An HTML Element reference guide

Developer Tools:

Google’s Intro to Dev Tools in Chrome

Firefox Dev Tools / Development Browser

CSS layouts and tools.

CSS boilerplate and layout examples.

CSS Boilerplates, Layouts etc.

CSS examples.

Many examples of advanced CSS design.

JavaScript and jQuery

Introductions and Reference:

Handy, straightforward.

Mozilla’s Developer Network reference for JavaScript (probably the best out there)

Slightly more advanced introduction, with some links to basics as well. Handy for concepts.
http://www.webmonkey.com/2010/02/get_started_with_jquery/
  ● Straightforward JQ tutorial.

https://oscarotero.com/jquery/
  ● jQuery quick reference

Examples:
http://timeline.knightlab.com/
  ● Interesting use of JavaScript and Google Docs to produce timelines.

  ● An interesting example of advanced css and jq. (Doesn’t work on all browsers.)

Libraries:
http://velocityjs.org/blast/
  ● Library for text handling

https://sbstjn.github.io/timesheet.js/
  ● Interesting, simple timeline creation.

On JQuery and SVG:
https://bigbitecreative.com/introduction-svg-animation/
  ● Learn a bit about manipulating SVG with jQuery, and how SVG works.

http://keith-wood.name/svgrref.html
  ● Nice SVG reference for the jQuery SVG plugin.

Misc.
http://www.regexpal.com/
  ● Resources for Regular Expressions

https://www.codecademy.com/learn/learn-git
  ● Codecademy tutorial to Git. (Free, with some advanced options for a charge.)

http://www.w3schools.com/
  ● Tutorials for many languages. These are hit-or-miss and don’t always give the best
  information, but the site does have some useful references and exercises, and the
  sandbox feature is nice.